conference

proceedings

The Sixth USENIX Security Symposium Proceedings

San Jose, California
July 22–25, 1996

Sponsored by
The USENIX Association
Co-sponsored by **UniForum in cooperation with the Computer Emergency Response Team (CERT)**

USENIX®
The UNIX® and Advanced Computing Systems Professional and Technical Association

USENIX The Sixth USENIX Security Symposium Proceedings San Jose, California July 1996

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: *office@usenix.org*
URL: *http://www.usenix.org*

The price is $27 for members and $35 for nonmembers.
Outside the U.S.A. and Canada, please add
$11 per copy for postage (via air printed matter).

Past USENIX UNIX Security Proceedings

Security V	June, 1995	Salt Lake City, Utah	$27/35
Security IV	October 1993	Santa Clara, CA	$15/20
Security III	September 1992	Baltimore, MD	$30/39
Security II	August 1990	Portland, OR	$13/16
Security	August 1988	Portland, OR	$7/7

ISBN 1-880446-79-0

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

Proceedings of the

Sixth Annual USENIX Security Symposium:

Focusing on Applications of Cryptography

July 22-25, 1996
San Jose, California

Table of Contents

6th USENIX Security Symposium
Focusing on Applications of Cryptography
July 22-25, 1996 ♦ San Jose, California

Wednesday, July 24

9am-10:30am

Opening Remarks: *Greg Rose, Qualcom*

Keynote Address: A Simple Distributed Security Infrastructure
Ronald L. Rivest, MIT Laboratory for Computer Science

11am-12:30pm

Session 1: ***"He who controls the area controls the religion"*****–Latin**

Session Chair: Brent Chapman, Great Circle Associates

2pm-3:30pm

Session 2: ***"The secret whispers of each other's watch"*** **–W.S.**

Session Chair: Avi Rubin, Bellcore

4pm-5:30pm

Session 3: ***"There are more things in heaven and earth"*****–W.S.**

Session Chair: Steven M. Bellovin, AT&T Research

Thursday, July 25

9am-10:30am — **Session 4: *"to promote commerce, and not betray it"–Pepys***
Session Chair: Clifford Neuman, University of Southern California

11am 12:30pm — **Session 5: *"In middle of her Web, which spreadeth wide"*–Davies**
Session Chair: Kathy Fithen, CERT

2pm-3:30pm — **Session 6: *"Two massy keys he bore"*–Milton**
Session Chair: Diane Coe, The MITRE Corporation

4pm-5pm — **Session 7: *"... tangled in amorous nets"*–Milton**
Session Chair: Fred Avolio, Trusted Information Systems, Inc.

ACKNOWLEDGMENTS

PROGRAM CHAIR

Greg Rose, *Qualcomm*

USENIX PROGRAM COMMITTEE

Fred Avolio, *Trusted Information Systems, Inc.*
Steven M. Bellovin, *AT&T Research*
Brent Chapman, *Great Circle Associates*
Diane Coe, *The MITRE Corporation*
Ed DeHart, *CERT*
Kathy Fithen, *CERT*
Daniel Geer, *Open Market, Inc.*
Peter Gutmann, *University of Auckland*
Kent Landfield, *Sterling Software*
Clifford Neuman, *University of Southern California*
Avi Rubin, *Bellcore*
Ken van Wyk, *Defense Information Systems Agency*
Karen Worstell, *The Boeing Company*

READERS

Matt Bishop, *University of California, Davis*
Lee Damon, *Qualcomm*
Phil Karn, *Qualcomm*
Greg Noel, *Qualcomm*
Brett Rees, *Sterling Software*

UNIFORUM PROGRAM CHAIR

Jim Schindler, *Hewlett-Packard*

UNIFORUM PROGRAM COMMITTEE

Rik Farrow, *Consultant*
Deborah Murray, *UniForum*

TUTORIAL COORDINATOR

Daniel V. Klein, *USENIX Association*

PROCEEDINGS PRODUCTION

Pennfield Jensen, *USENIX Association*

USENIX MEETING PLANNER

Judith F. Desharnais, *USENIX Association*

USENIX EXECUTIVE DIRECTOR

Ellie Young, *USENIX Association*

USENIX Marketing Director

Zanna Knight, *USENIX Association*

USENIX SUPPORT STAFF

Colleen Biddle, *USENIX Association*
Eileen Curtis, *USENIX Association*
Diane DeMartini, *USENIX Association*
Toni Veglia, *USENIX Association*

Preface

Since the last USENIX Security Symposium, just a year ago, the world has really awakened to the Internet, sat up and looked around, and started to get worried. Laws have been proposed and some passed, mechanisms for security and censorship designed, secure products broken, and censors bypassed.

Along with the growing awareness that security is important has come a realization that there are some problems only cryptographic techniques will solve. While great advances in cryptography have been made in the last two decades, deployment of these advances has been slow. It is time to correct that, and by focusing this year on the subject of Applications of Cryptography, this symposium attempts to contribute materially to the real world in this area.

Just after I drafted the preceding paragraph, I was on a United Airlines flight, and the in-flight news program had two interesting stories which are relevant to this conference. The first was about Tsutomu Shimomura's tracking of Kevin Mitnick, and it highlighted (among other things) many problems with the state of security in today's world, both on the Internet and in the telephone system. While one screen of email scrolled past Tsutomu's reflected face, I saw a number of familiar names, Blaze, Rubin, Bellovin, Karn, Cheswick, all of whom have contributed materially to this Symposium. So have a lot of other people, and I heartily thank all of them.

The second was about Football. I was dozing through it when I awoke with a start. The football coach was transmitting instructions to the quarterback (please forgive me if I get the terms wrong, I come from a country with different kinds of football, and I ignore them too...) by radio to a receiver in the player's helmet. He said that the information was encrypted so that spectators, and presumably the opposing coach, couldn't listen in, and that it would take years to crack the "very long" key. While I'm prepared to make a small wager that "very long" key was 40 bits, the fact that cryptography is being deployed on football fields makes my point.

I hope, and believe, that these proceedings and the Symposium itself have been valuable for the attendees and the broader community.

Good luck,

Greg Rose
Program Chair

A Secure Environment for Untrusted Helper Applications

Confining the Wily Hacker

Ian Goldberg David Wagner Randi Thomas Eric A. Brewer
{iang,daw,randit,brewer}@cs.berkeley.edu
University of California, Berkeley

Abstract

Many popular programs, such as Netscape, use untrusted helper applications to process data from the network. Unfortunately, the unauthenticated network data they interpret could well have been created by an adversary, and the helper applications are usually too complex to be bug-free. This raises significant security concerns. Therefore, it is desirable to create a secure environment to contain untrusted helper applications. We propose to reduce the risk of a security breach by restricting the program's access to the operating system. In particular, we intercept and filter dangerous system calls via the Solaris process tracing facility. This enabled us to build a simple, clean, user-mode implementation of a secure environment for untrusted helper applications. Our implementation has negligible performance impact, and can protect pre-existing applications.

1 Introduction

Over the past several years the Internet environment has changed drastically. This network, which was once populated almost exclusively by cooperating researchers who shared trusted software and data, is now inhabited by a much larger and more diverse group that includes pranksters, crackers, and business competitors. Since the software and data exchanged on the Internet is very often unauthenticated, it could easily have been created by an adversary.

Web browsers are an increasingly popular tool for retrieving data from the Internet. They often rely on helper applications to process various kinds of information. These helper applications are security-critical, as they handle untrusted data, but they are not particularly trustworthy themselves. Older versions of `ghostscript`, for example, allowed malicious programs to spawn processes and to read or write an unsuspecting user's files [15, 18, 19, 34, 36]. What is needed in this new environment, then, is protection for all resources on a user's system from this threat.

Our aim is to confine the untrusted software and data by monitoring and restricting the system calls it performs. We built Janus[1], a secure environment for untrusted helper applications, by taking advantage of the Solaris process tracing facility. Our primary goals for the prototype implementation include security, versatility, and configurability. Our prototype is meant to serve as a proof-of-concept, and we believe our techniques may have a wider application.

2 Motivation

2.1 The threat model

Before we can discuss possible approaches to the problem, we need to start by clarifying the threat model. Web browsers and `.mailcap` files make it convenient for users to view information in a wide variety of formats by de-multiplexing documents to helper applications based on the document format. For example, when a user downloads a Postscript document from a remote network site, it may be automatically handled by `ghostview`. Since that downloaded data could be under adversarial control, it is completely untrustworthy. We are concerned that an adversary could send malicious data that subverts the document viewer (through some unspecified security bug or misfeature), compromising the user's security. Therefore we consider helper applications untrusted, and wish to place them outside the host's trust perimeter.

[1] Janus is the Roman god of entrances and exits, who had two heads and eternally kept watch over doorways and gateways to keep out intruders.

We believe that this is a prudent level of paranoia. Many helper programs were initially envisioned as a viewer for a friendly user and were not designed with adversarial inputs in mind. Furthermore, `ghostscript` implements a full programming language, with complete access to the filesystem; many other helper applications are also very general. Worse still, these programs are generally big and bloated, and large complex programs are notoriously insecure.[2] Security vulnerabilities have been exposed in these applications [15, 18, 19, 34, 36].

2.2 The difficulties

What security requirements are demanded from a successful protection mechanism? Simply put, an outsider who has control over the helper application must not be able to compromise the confidentiality, integrity, or availability of the rest of the system, including the user's files or account. Any damage must be limited to the helper application's display window, temporary files and storage, and associated short-lived objects. In other words, we insist on the Principle of Least Privilege: the helper application should be granted the most restrictive collection of capabilities required to perform its legitimate duties, and no more. This ensures that the damage a compromised application can cause is limited by the restricted environment in which it executes. In contrast, an unprotected Unix application that is compromised will have all the privileges of the account from which it is running, which is unacceptable.

Imposing a restricted execution environment on helper applications is more difficult than it might seem. Many traditional paradigms such as the reference monitor and network firewall are insufficient on their own, as discussed below. In order to demonstrate the difficulty of this problem and appreciate the need for a novel solution, we explore several possible approaches.

BUILDING SECURITY DIRECTLY INTO EACH HELPER APPLICATION: Taking things to the extreme, we could insist all helper applications be rewritten in a simple, secure form. We reject this as completely unrealistic; it is simply too much work to re-implement them. More practically, we could adopt a reactive philosophy, recognizing individual weaknesses as each appears and engineering security patches one at a time. Historically, this has been a losing battle, at least for large applications: for instance, explore the sad tale of the `sendmail` "bug of the month" [1, 2, 3, 4, 8, 9, 10, 11, 12, 13, 14, 16]. In any event, attempts to build security directly into the many helper applications would require each program to be considered separately—not an easy approach to get right. For now, we are stuck with many useful programs which offer only minimal assurances of security; therefore what we require is a general, external protection mechanism.

ADDING NEW PROTECTION FEATURES INTO THE OS: We reject this design for several reasons. First, it is inconvenient. Development and installation both require modifications to the kernel. This approach, therefore, has little chance of becoming widely used in practice. Second, wary users may wish to protect themselves without needing the assistance of a system administrator to patch and recompile the operating system. Third, security-critical kernel modifications are very risky: a bug could end up allowing new remote attacks or allow a compromised application to subvert the entire system. The chances of exacerbating the current situation are too high. Better to find a user-level mechanism so that users can protect themselves, and so that pre-existing access controls can serve as a backup; even in the worst case, security cannot decrease.

THE PRE-EXISTING REFERENCE MONITOR: The traditional operating system's monolithic reference monitor cannot protect against attacks on helper applications directly. At most, it could prevent a penetration from spreading to new accounts once the browser user's account has been compromised, but by then the damage has already been done. In practice, against a motivated attacker most operating systems fail to prevent the spread of penetration; once one account has been subverted, the whole system typically falls in rapid succession.

THE CONVENTIONAL NETWORK FIREWALL: Packet filters cannot distinguish between different types of HTTP traffic, let alone analyze the data for security threats. A proxy could, but it would be hard-pressed to understand all possible file formats, interpret the often-complex application languages, and squelch all dangerous data. This would make for a very complex and thus untrustworthy proxy.

We therefore see the need for a new, simple, and general user-level protection mechanism that does not require modification of existing helper applications or operating systems. The usual techniques and conventional paradigms do not work well in this situation. We hope that the difficulty of the problem

[2] For instance, `ghostscript` is more than 60,000 lines of C; and `mpeg_play` is more than 20,000 lines long.

and the potential utility of a solution should help to motivate interest in our project.

3 Design

Our design, in the style of a reference monitor, centers around the following basic assumption:

> AN APPLICATION CAN DO LITTLE HARM IF ITS ACCESS TO THE UNDERLYING OPERATING SYSTEM IS APPROPRIATELY RESTRICTED.

Our goal, then, was to design a user-level mechanism that monitors an untrusted application and disallows harmful system calls.

A corollary of the assumption is that an application may be allowed to do anything it likes that does not involve a system call. This means it may have complete access to its address space, both code and data. Therefore, any user-level mechanism we provide must reside in a different address space. Under Unix, this means having a separate process.

One of our basic design goals was SECURITY. The untrusted application should not be able to access any part of the system or network for which our program has not granted it permission. We use the term *sandboxing* to describe the concept of confining a helper application to a restricted environment, within which it has free reign. This term was first introduced, in a slightly different setting, in [35].

To achieve security, a slogan we kept in mind was "keep it simple" [29]. Simple programs are more likely to be secure; simplicity helps to avoid bugs, and makes it easier to find those which creep in [17, Theorem 1]. We would like to keep our program simpler than the applications that would run under it.

Another of our goals was VERSATILITY. We would like to be able to allow or deny individual system calls flexibly, perhaps depending on the arguments to the call. For example, the **open** system call could be allowed or denied depending on which file the application was trying to open, and whether it was for reading or for writing.

Our third goal was CONFIGURABILITY. Different sites have different requirements as to which files the application should have access, or to which hosts it should be allowed to open a TCP connection. In fact, our program ought to be configurable in this way even on a per-user or per-application basis.

On the other hand, we did *not* strive for the criteria of safety or portability of applications. By *safety*, we mean protecting the application from its own bugs. We allow the user to run any program he wishes, and we allow the executable to play within its own address space as much as it would like.

We adopted for our program, then, a simple, modular design:

- a *framework*, which is the essential body of the program, and
- dynamic *modules*, used to implement various aspects of a configurable security policy by filtering relevant system calls.

The framework reads a configuration file, which can be site-, user-, or application-dependent. This file lists which of the modules should be loaded, and may supply parameters to them. For example, the configuration line

```
path allow read,write /tmp/*
```

would load the `path` module, passing it the parameters "`allow read,write /tmp/*`" at initialization time. This syntax is intended to allow files under `/tmp` to be opened for reading or writing.

Each module filters out certain dangerous system call invocations, according to its area of specialization. When the application attempts a system call, the framework dispatches that information to relevant policy modules. Each module reports its opinion on whether the system call should be permitted or quashed, and any necessary action is taken by the framework. We note that, following the Principle of Least Privilege, we let the operating system execute a system call only if some module explicitly allows it; the default is for system calls to be denied. This behavior is important because it causes the system to err on the side of security in case of an underspecified security policy.

Each module contains a list of system calls that it will examine and filter. Note that some system calls may appear in several modules' lists. A module may assign to each system call a function which validates the arguments of the call *before* the call is executed

by the operating system.[3] The function can then use this information to optionally update local state, and then suggest allowing the system call, suggest denying it, or make no comment on the attempted system call.

The suggestion to allow is used to indicate a module's explicit approval of the execution of this system call. The suggestion to deny indicates a system call which is to be denied execution. Finally, a "no comment" response means that the module has no input as to the dispatch of this system call.

Modules are listed in the configuration file from most general to most specific, so that the last relevant module for any system call dictates whether the call is to be allowed or denied. For example, a suggestion to allow countermands an earlier denial. Note that a "no comment" response has no effect: in particular, it does not override an earlier "deny" or "allow" response.

Normally, when conflicts arise, earlier modules are overridden by later ones. To escape this behavior, for very special circumstances modules may unequivocally allow or deny a system call and explicitly insist that their judgement be considered final. In this case, no further modules are consulted; a "super-allow" or "super-deny" cannot be overridden. The intent is that this feature should be used quite rarely, for only the most critical of uses. Write access to `.rhosts` could be super-denied near the top of the configuration file, for example, to provide a safety net in case we accidentally miswrite a subsequent file access rule.

In designing the framework we aimed for simplicity and versatility as much as possible, though these goals often conflict. One can imagine more versatile and sophisticated algorithms to dispatch system calls, but they would come at a great cost to simplicity.

4 Implementation

4.1 Choice of operating system

In order to implement our design, we needed to find an operating system that allowed one user-level process to watch the system calls executed by another process, and to control the second process in various ways (such as causing selected system calls to fail).

Luckily, most operating systems have a process-tracing facility, intended for debugging. Most operating systems offer a program called `trace(1)`, `strace(1)`, or `truss(1)` which can observe the system calls performed by another process as well as their return values. This is often implemented with a special system call. `ptrace(2)`, which allows the tracer to register a callback that is executed whenever the tracee issues a system call. Unfortunately, `ptrace` offers only very coarse-grained all-or-nothing tracing: we cannot trace a few system calls without tracing all the rest as well. Another disadvantage of the `ptrace(2)` interface is that many OS implementations provide no way for a tracing process to abort a system call without killing the traced process entirely.

Some more modern operating systems, such as Solaris 2.4 and OSF/1, however, offer a better process-tracing facility through the `/proc` virtual filesystem. This interface allows direct control of the traced process's memory. Furthermore, it has fine-grained control: we can request callbacks on a per-system call basis.

There are only slight differences between the Solaris and the OSF/1 interfaces to the `/proc` facility. One of them is that Solaris provides an easy way for the tracing process to determine the arguments and return values of a system call performed by the traced process. Also, Solaris operating system is somewhat more widely deployed. For these reasons, we chose Solaris 2.4 for our implementation.

4.2 The policy modules

4.2.1 Overview

The policy modules are used to select and implement security policy decisions. They are dynamically loaded at runtime, so that different security policies can be configured for different sites, users, or applications. We implemented a sample set of modules that can be used to set up the traced application's environment, and to restrict its ability to read or write files, execute programs, and establish TCP connections. In addition, the traced application is prevented from performing certain system calls, as described below. The provided modules offer considerable flexibility themselves, so that may configure them simply by editing their parameters in the configuration file. However, if different modules

[3] In addition, a module can assign to a system call a similar function which gets called *after* the system call has executed, just before control is returned to the helper application. This function can examine the arguments to the system call, as well as the return value, and update the module's local state.

are desired or required, it is very simple to compile new ones.

Policy modules need to make a decision as to which system calls to allow, which to deny, and for which a function must be called to determine what to do. The first two types of system calls are the easiest to handle.

Some examples of system calls that are always allowed (in our sample modules) are `close`, `exit`, `fork`, and `read`. The operating system's protection on these system calls is sufficient for our needs.

Some examples of system calls that are always denied (in our sample modules) are ones that would not succeed for an unprivileged process anyway, like `setuid` and `mount`, along with some others, like `chdir`, that we disallow as part of our security policy.

The hardest system calls to handle are those for which a function must, in general, be called to determine whether the system call should be allowed or denied. The majority of these are system calls such as `open`, `rename`, `stat`, and `kill` whose arguments must be checked against the configurable security policy specified in the parameters given to the module at load time.

4.2.2 Sample security policy

We implemented a sample security policy to test our ideas, as a proof of concept.

Helper applications are allowed to `fork` children, we then recursively trace. Traced processes can only send signals to themselves or to their children, and never to an untraced application. Environment variables are initially sanitized, and resource usage is carefully limited.

In our policy, access to the filesystem is severely limited. A helper application is placed in a particular directory; it cannot `chdir` out of this directory. We allow it full access to files in or below this directory; to prevent escape from this sandbox directory, access to paths containing `..` are always denied. The untrusted application is allowed read access to certain carefully controlled files referenced by absolute pathnames, such as shared libraries and global configuration files. We concentrate all access control in the `open` system call, and always allow `read` and `write` calls; this is safe, because `write` is only useful when used on a file descriptor obtained from a system call like `open`. This approach simplifies matters, and also allows us a performance optimization further down the line; see Section 4.4.

Of course, protecting the filesystem alone is not enough. Nearly any practical helper application will require access to network resources. For example, all of the programs we considered need to open a window on the X11 display to present document contents. In our security policy, network access must be carefully controlled: we allow network connections only to the X display, and this access is allowed only through a safe X proxy.

X11 does not itself provide the security services we require (X access control is all-or-nothing). A rogue X client has full access to all other clients on the same server, so an otherwise confined helper application could compromise other applications if it were allowed uncontrolled access to X. Fortunately the firewall community has already built several safe X proxies that understand the X protocol and filter out dangerous requests [26, 31]. We integrated our Janus prototype with `Xnest` [31], which lets us run another complete instance of the X protocol under `Xnest`. `Xnest` acts as a server to its clients (e.g. untrusted helper applications), but its display is painted within one window managed by the root X server. In this way, untrusted applications are securely encapsulated within the child `Xnest` server and cannot escape from this sandbox display area or affect other normal trusted applications. `Xnest` is not ideal—it is not as small or simple as we would like—but further advances in X protocol filtering are likely to improve the situation.

4.2.3 Sample modules

Our modules implementing this sample policy are as follows. The `basic` module supplies defaults for the system calls which are easiest to analyze, and takes no configuration parameters. The `putenv` module allows one to specify environment variable settings for the traced application via its parameters; those which are not explicitly mentioned are unset. The special parameter `display` causes the helper application to inherit the parent's `DISPLAY`. The `tcpconnect` module allows us to restrict TCP connections by host and/or port; the default is to disallow all connections. The `path` module, the most complicated one, lets one allow or deny file accesses according to one or more patterns.

Because this policy is just an example, we have not gone into excruciating detail regarding the specific policy decisions implemented in our modules.

Our sample configuration file for this policy can be seen in Figure 2 in the Appendix.

4.3 The framework

4.3.1 Reading the configuration file

The framework starts by reading the configuration file, the location of which can be specified on the command line. This configuration file consists of lines like those shown in Figure 2: the first word is the name of the module to load, and the rest of the line acts as a parameter to the module.

For each module specified in the configuration file, `dlopen(3x)` is used to dynamically load the module into the framework's address space. The module's `init()` function is called, if present, with the parameters for the module as its argument.

The list of system calls and associated values and functions in the module is then merged into the framework's *dispatch table*. The dispatch table is an array, indexed by system call number, of linked lists. Each value and function in the module is appended to the list in the dispatch table that is indexed by the system call to which it is associated.

The result, after the entire configuration file has been read, is that for each system call, the dispatch table provides a linked list that can be traversed to decide whether to allow or deny a system call.

4.3.2 Setting up the traced process

After the dispatch table is set up, the framework gets ready to run the application that is to be traced: a child process is `fork()`ed, and the child's state is cleaned up. This includes setting a `umask` of 077, setting limits on virtual memory use, disabling core dumps, switching to a sandbox directory, and closing unnecessary file descriptors. Modules get a chance to further initialize the child's state; for instance, the `putenv` module sanitizes the environment variables. The parent process waits for the child to complete this cleanup, and begins to debug the child via the `/proc` interface. It sets the child process to stop whenever it begins or finishes a system call (actually, only a subset of the system calls are marked in this manner; see Section 4.4, below). The child waits until it is being traced, and executes the desired application.

In our sample security policy, the application is confined to a sandbox directory. By default, this directory is created in `/tmp` with a random name, but the `SANDBOX_DIR` environment variable can be used to override this choice.

4.3.3 Running the traced process

The application runs until it performs a system call. At this point, it is put to sleep, and the tracing process wakes up. The tracing process determines which system call was attempted, along with the arguments to the call. It then traverses the appropriate linked list in the dispatch table, in order to determine whether to allow or to deny this system call.

If the system call is to be allowed, the tracing process simply wakes up the application, which proceeds to complete the system call. If, however, the system call is to be denied, the tracing process wakes up the application with the `PRSABORT` flag set. This causes the system call to abort immediately, returning a value indicating that the system call failed and setting `errno` to `EINTR`. In either case, the tracing process goes back to sleep.

The fact that an aborted system call returns `EINTR` to the application presents a potential problem. Some applications are coded in such a way that, if they receive an `EINTR` error from a system call, they will retry the system call. Thus, if such a application tries to execute a system call which is denied by the security policy, it will get stuck in a retry loop. We detect this problem by noticing when a large number (currently 100) of the same system call with the same arguments are consecutively denied. If this occurs, we assume the traced application is not going to make any further progress, and just kill the application entirely, giving an explanatory message to the user. We would prefer to be able to return other error codes (such as `EPERM`) to the application, but Solaris does not support that behavior.

When a system call completes, the tracing process has the ability to examine the return value if it so wishes. If any module had assigned a function to be executed when this system call completes, as described above, it is executed at this time. This facility is not widely used, except in one special case.

When a `fork()` or `vfork()` system call completes, the tracing process checks the return value and then `fork()`s itself. The child of the tracing process then detaches from the application, and begins tracing the application's child. This method safely allows the traced application to spawn a child (as `ghostview`

spawns `gs`, for example) by ensuring that all children of untrusted applications are traced as well.

We have not aimed for extensive auditing, but logging of the actions taken by the framework would be easy to add to our implementation if desired.

We should point out that the Solaris tracing facilities will not allow a traced application to `exec()` a setuid program. Furthermore, traced programs cannot turn off their own tracing.

4.4 The optimizer

Our program has the potential to add a non-trivial amount of overhead to the traced application whenever it intercepts a system call. In order to keep this overhead down, we obviously want to intercept as few system calls as possible—or at least, as few of the common ones as possible. However, we do not wish to give up security to gain performance.

Therefore, we apply several optimizations to the system call dispatch table before the untrusted helper application executes. We note that one common case arises when a module's system call handler always returns the same allow/deny value (and leaves no side effects); this special case allows us to remove redundant values in the dispatch table.

The most important optimization observes that certain system calls, such as `write`, are always allowed; so we need not register a callback with the OS for them. This avoids the extra context switches to and from the tracing process each time the traced application makes such a system call, and thus those system calls can execute at full speed as though there were no tracing or filtering. By eliminating the need to trace common system calls such as `read` and `write`, we can greatly speed up the common case.

5 Evaluation

The general population is more interested in efficiency and convenience than in security, so any security product intended for general use must address these concerns. For this reason, we evaluate our prototype implementation by a number of criteria, including security, applicability, and ease of use, in addition to performance.

5.1 Ease of use

The secure environment is relatively easy to install. All that is needed is to protect the invocation of any helper application with our environment. The most convenient solution is to specify our `janus` program in a `mailcap` file, which could look like

```
image/*; janus xv %s
application/postscript; janus ghostview %s
video/mpeg; janus mpeg_play %s
video/*; janus xanim %s
```

With little effort, a system administrator could set up the in-house security policy by listing `janus` in the default global `mailcap` file; then the secure environment would be transparent to all the users on the system. Similarly, users could protect themselves by doing the same to their personal `.mailcap` file.

5.2 Applicability

Users will want to run our secure environment with pre-existing helper applications. We tested a number of programs under our secure environment, including `ghostview`, `mpeg_play`, `xdvi`, `xv`, and `xanim`. Though we followed the Principle of Least Privilege and were very restrictive in our security policy, we found that each of the applications had sufficient privilege, and we had not unduly restricted the applications from doing their legitimate intended jobs.

In addition, we ran the shells `sh` and `bash` under our secure environment. Unless the user explicitly tries to violate the security policy (e.g., by writing to `.rhosts`), there is no indication of the restricted nature of the shell. Attempts to violate the security policy are rewarded with a shell error message.

5.3 Security

There is no universally accepted way to assess whether our implementation is secure; however, there are definite indications we can use to make this decision.

We believe in security through simplicity, and this was a guiding principle throughout the design and implementation process. Our entire implementation consists of approximately 2100 lines of code: the framework has 800, and the modules have the remaining 1300. Furthermore, we have attempted to minimize the amount of security-critical state where

possible. Since the design concept is a simple one, and because the entire program is small, the implementation is easier to understand and to evaluate. Thus, there is a much smaller chance of having an undetected security hole.

We performed some simple sanity checks to verify that our implementation appropriately restricts applications. More work on assurance is needed.

Most importantly, the best test is outside scrutiny by independent experienced security researchers; a detailed code review would help improve the assurance and security offered by our secure environment. All are encouraged to examine our implementation for flaws.

5.4 Performance

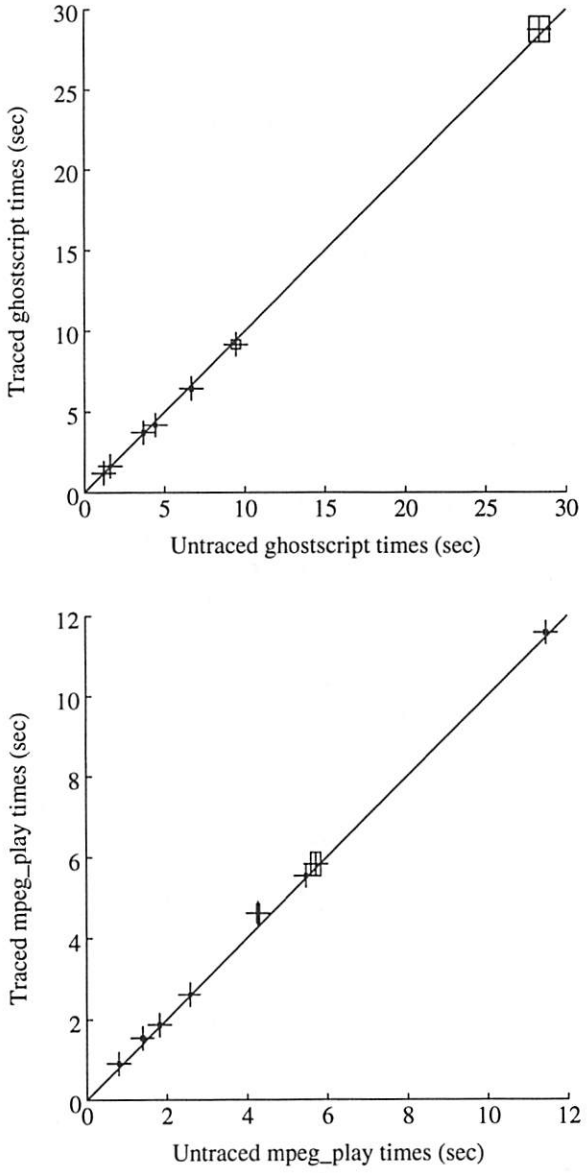
Since our design potentially adds time-consuming context switches for every system call the untrusted application makes, the obvious performance metric to evaluate is time. We measured the peak performance of `ghostscript` and `mpeg_play`, two large commonly used helper applications, under our secure environment. Note that `mpeg_play` in particular is performance critical.

`mpeg_play` was used to display nine mpeg movies ranging in size from 53 KB (18 frames) to 740 KB (400 frames). `ghostscript` was used to display seven Postscript files ranging in size from 9 KB to 1.7 MB. `ghostscript` was run non-interactively, so that all the pages in the Postscript file were displayed in succession with no user intervention. We took 100 measurements for each file, 50 traced under our secure environment and 50 untraced, calculating the mean and standard deviation for each set. The measurements were done using an unloaded single-processor SPARCstation 20 workstation running Solaris 2.4. The `Xnest` X windows proxy [31] was used with the secure environment, but not with the untraced measurements.

The results are displayed in Figure 1. For each set, we plotted the traced time against the untraced time.[4] The boxes around the data points indicate one standard deviation. The diagonal line shows the ideal result of no statistically significant performance overhead. In the best possible case, the error boxes will all intersect the ideal line. Boxes entirely above the line indicate statistically significant overhead. As can be seen, the secure environment imposes no significant performance penalty.

[4] Similar results were obtained when measuring frame rate per second for `mpeg_play`.

Figure 1: Performance data for `ghostscript` and `mpeg_play`



The negligible performance impact can be attributed to the unintrusive nature of our implementation. Of course, all computations and memory references that do not involve the OS will execute at full speed, so system calls can be the only source of performance overhead. We first note that system calls are already so time-consuming that the additional overhead of the Janus process filtering is insignificant. Furthermore, most of the heavily used system calls (such as `read` and `write`) require no access checks and

therefore run at full speed. By staying out of the application's way and optimizing for the common case, we have allowed typical applications to run with negligible performance overhead.

6 Related work

Due to the accelerated development of communication technology, the security and protection problems inherent in an open and free communication environment, such as the Internet, are relatively new ones to solve. Consequently, much of the work addressing security for this environment is still being developed.

To achieve security, we use the concept of sandboxing, first introduced by Wahbe et al. in the context of software fault isolation [35]. However, they were actually solving a different problem. What they achieved was safety for trusted modules running in the same address space as untrusted modules. They ignored the problem of system-level security; conversely, we do not attempt to provide safety. They also use binary-rewriting technology to accomplish their goals, which prevents them from running arbitrarily general pre-existing applications.

Java [25] is an comprehensive system that addresses, among other things, both safety and security, although it achieves security by a different approach from ours. Java cannot secure pre-existing programs, because it requires use of a new language. We do not have this problem; our design will run any application, and so is more versatile in this respect. However, Java offers many other advantages that we do not address; for instance, Java provides architecture-independence, while Janus only applies to native code and provides no help with portability.

OmniWare [20] takes advantage of software fault isolation techniques and compiler support to safely execute untrusted code. Like Java, it also has architecture-independence, extensibility, and efficiency as important goals.

We note two important differences between the Java approach and the Janus philosophy. The Java protection mechanism is much more complex, and is closely intertwined with the rest of Java's other functionality. In contrast, we have more limited goals, explicitly aim for extreme simplicity, and keep the security mechanism orthogonal from the non-security-critical functionality.

`securelib` is a shared library that replaces the C `accept`, `recvfrom`, and `recvmsg` library calls by a version that performs address-based authentication; it is intended to protect security-critical Unix system daemons [30]. Other research that also takes advantage of shared libraries can be found in [27, 24]. We note that simple replacement of dangerous C library calls with a safe wrapper is insufficient in our extended context of untrusted and possibly hostile applications; a hostile application could bypass this access control by simply issuing the dangerous system call directly without invoking any library calls.

Fernandez and Allen [23] extend the filesystem protection mechanism with per-user access control lists. Lai and Gray [28] describe an approach which protects against Trojan horses and viruses by limiting filesystem access: their OS extension confines user processes to the minimal filesystem privileges needed, relying on hints from the command line and (when necessary) run-time user input. TRON [7] discourages Trojan horses by adding per-process capabilities support to the filesystem discretionary access controls. These works all suffer two major disadvantages: they require kernel modifications, and they do not address issues such as control over process and network resources.

Domain and Type Enforcement (DTE) is a way to extend the OS protection mechanisms to let system administrators specify fine-grained mandatory access controls over the interaction between security-relevant subjects and objects. A research group at TIS has amassed considerable experience with DTE and its practical application to Unix systems [5, 6, 32, 33]. DTE is an attractive and broadly applicable approach to mandatory access control, but its main disadvantage is that it requires kernel modifications; we aimed instead for user-level protection.

7 Limitations and future work

7.1 Limitations of the prototype

One inherent limitation of the Janus implementation is that we can only successfully run helper applications which do not legitimately need many privileges. Our approach will easily accommodate any program that only requires simple privileges, such as access to a preferences file. Application developers may want to keep this in mind and not assume, for example, that their applications will be able to access the whole filesystem.

We have followed one simple direction in our prototype implementation, but others are possible as well.

One could consider using specialized Unix system calls to revoke certain privileges. The two major contenders are `chroot()`, to confine the application within a safe directory structure, and `setuid()`, to change to a limited-privilege account such as `nobody`. Unfortunately, programs need superuser privileges to use these features; since we were committed to a user-level implementation, we decided to ignore them. However, this design choice could be reconsidered. Other security policies (such as mandatory audit logs) may also be more appropriate in some environments.

The most fundamental limitation of our implementation, however, stems from its specialization for a single operating system. Each OS to which Janus might be ported requires a separate security analysis of its system calls. Also, a basic assumption of Janus is that the operating system provides multiple address spaces, allows trapping of system calls, and makes it feasible to interpose proxies where necessary. Solaris 2.4 has the most convenient support for these mechanisms; we believe our approach may also apply to some other Unix systems. On the other hand, platforms that do not support these services cannot directly benefit from our techniques. In particular, our approach cannot be applied to PCs running MS-DOS or Microsoft Windows. The utility of these confinement techniques, then, will be determined by the underlying operating system's support for user-level security primitives.

7.2 Future work

In this paper, we have limited discussion to the topic of protecting untrusted helper applications. It would also be interesting to explore how these techniques might be extended to a more ambitious scope.

One exciting area for further research involves Java applet security. Java [25] is seeing widespread deployment, but several implementation bugs [22] have started to shake confidence in its security model. For more protection, one could run Java applets within a secure environment built from techniques described in this paper. This approach provides defense in depth: if the Java applet security mechanism is compromised, there is still a second line of defense. We are experimenting with this approach; more work is needed.

Another natural extension of this work is to run web browsers under the Janus secure environment. The recursive tracing of child processes would ensure that running a browser under Janus would protect all spawned helper applications as well. The arguments which leave us suspicious of helper applications also apply to web browsers: they are large, complex programs that interpret untrusted network data. For example, a buffer overrun bug was found in an earlier version of the Netscape browser [21]. The main challenge is that browsers legitimately require many more privileges; for instance, most manage configuration files, data caches, and network connections. Of these, the broader network access seems to pose the most difficulties.

We believe proxies are a promising approach for improving control over network accesses. By taking advantage of earlier work in firewalls, we were able to easily integrate a safe X proxy into our prototype. We have shown that one can guard access to system calls with a reference monitor constructed from process-tracing facilities; we suspect that one can effectively and flexibly guard access to the outside network with existing proxies developed by the firewall community. One issue is how to interpose proxies forcibly upon untrusted and uncooperative applications. We currently use environment variables as hints—for instance, we change the `DISPLAY` variable to point to a proxy X server, and disallow access to any other X display—but this only works for well-behaved applications that consult environment variables consistently. One might consider implementing such hints with a shared library that replaces network library calls with a safe call to a secure proxy.

So far we have followed the policy that a helper application should not be able to communicate with the outside network, since there are several subtle security issues with address-based authentication, trust perimeters, and covert channels [22]. Integration with filtering proxies and fine-grained control over access to other network services, such as domain nameservers and remote web servers, would enable our techniques to be used in broader contexts. The overlap with research into firewalls lends hope that these problems can be solved satisfactorily.

8 Conclusion

We designed and implemented a secure environment for untrusted helper applications. We restrict an untrusted program's access to the operating system by using the process tracing facility available in Solaris 2.4. In this way, we have demonstrated the feasibility of building and enforcing practical security for untrusted helper applications.

The Janus approach has two main advantages:

- The Janus protection mechanism is *orthogonal* from other application functionality, so our user-mode implementation is simple and clean. This makes it more likely to be secure, and allows our approach to be broadly applicable to all applications.
- We can protect existing applications with little performance penalty.

We feel that this effort is a valuable step toward security for the World Wide Web.

9 Availability

The Web page
`http://www.cs.berkeley.edu/~daw/janus/`
will contain more information on availability of the Janus software described in this paper.

10 Acknowledgements

We would like to thank Steven Bellovin, David Oppenheimer, Armando Fox, Steve Gribble, and the anonymous reviewers for their helpful comments.

References

[1] [8lgm]-Advisory-16.UNIX.sendmail-6-Dec-1994, December 1994.

[2] [8lgm]-Advisory-17.UNIX.sendmailV5-2-May-1995, May 1995.

[3] [8lgm]-Advisory-17.UNIX.sendmailV5.22-Aug-1995, August 1995.

[4] [8lgm]-Advisory-20.UNIX.sendmailV5.1-Aug-1995, August 1995.

[5] Lee Badger, Daniel F. Sterne, David L. Sherman, and Kenneth M. Walker. A domain and type enforcement UNIX prototype. *USENIX Computing Systems*, 9(1):47–83, Winter 1996.

[6] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. Practical domain and type enforcement for UNIX. In *Proc. 1995 IEEE Symposium on Security and Privacy*, 1995.

[7] Andrew Berman, Virgil Bourassa, and Erik Selberg. TRON: Process-specific file protection for the UNIX operating system. In *Proc. 1995 USENIX Winter Technical Conference*, pages 165–175. USENIX Assoc., 1995.

[8] CERT advisory CA-88:01, 1988.

[9] CERT advisory CA-90:01, January 1990.

[10] CERT advisory CA-93:15, October 1993.

[11] CERT advisory CA-93:16, November 1993.

[12] CERT advisory CA-94:12, July 1994.

[13] CERT advisory CA-95:05, February 1995.

[14] CERT advisory CA-95:08, August 1995.

[15] CERT advisory CA-95:10, August 1995.

[16] CERT advisory CA-95:11, September 1995.

[17] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, 1994.

[18] Frederick Cohen. Personal communication.

[19] Frederick Cohen. Internet holes. *Network Security Magazine*, January 1996.

[20] Colusa Software. OmniWare technical overview, 1995.

[21] Ray Cromwell. Buffer overflow, September 1995. Announced on the Internet. `http://www.c2.net/hacknetscape/`.

[22] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java security: From HotJava to Netscape and beyond. In *Proc. of the 1996 IEEE Symposium on Security and Privacy*, 1996.

[23] G. Fernandez and L. Allen. Extending the Unix protection model with access control lists. In *Proc. Summer 1988 USENIX Conference*, pages 119–132. USENIX Assoc., 1988.

[24] Glenn S. Fowler, Yennun Huang, David G. Korn, and Herman Rao. A user-level replicated file system. In *Summer 1993 USENIX Conference Proceedings*, pages 279–290. USENIX Assoc., 1993.

[25] James Gosling and Henry McGilton. The Java language environment: A white paper, 1995. `http://www.javasoft.com/whitePaper/javawhitepaper_1.html`.

[26] Brian L. Kahn. Safe use of X window system protocol across a firewall. In *Proc. of the 5th USENIX UNIX Security Symposium*, 1995.

[27] David G. Korn and Eduardo Krell. The 3-D file system. In *Summer 1989 USENIX Conference Proceedings*, pages 147–156. USENIX Assoc., 1989.

[28] Nick Lai and Terence Gray. Strengthening discretionary access controls to inhibit Trojan horses and computer viruses. In *Proc. Summer 1988 USENIX Conference*, pages 275–286. USENIX Assoc., 1988.

[29] Butler Lampson. Hints for computer system design. In *Proceedings of the 9th ACM Symposium on Operating Systems Review*, volume 17:5, pages 33–48. Bretton Woods, 1983.

[30] William LeFebvre. Restricting network access to system daemons under SunOS. In *UNIX Security Symposium III Proceedings*, pages 93–103. USENIX Assoc., 1992.

[31] Davor Matic. Xnest. Available in the X11R6 source. Also `ftp://ftp.cs.umass.edu/pub/rcf/exp/X11R6/xc/programs/Xserver/hw/xnest`.

[32] David L. Sherman, Daniel F. Sterne, Lee Badger, and S. Murphy. Controlling network communication with domain and type enforcement. Technical Report 523, TIS, March 1995.

[33] Daniel F. Sterne, Terry V. Benzel, Lee Badger, Kenneth M. Walker, Karen A. Oostendorp, David L. Sherman, and Michael J. Petkac. Browsing the web safely with domain and type enforcement. In *1996 IEEE Symposium on Security and Privacy*, 1996. Research abstract.

[34] Jeff Uphoff. Re: Guidelines on cgi-bin scripts, August 1995. Post to `bugtraq` mailing list. `http://www.eecs.nwu.edu/cgi-bin/mfs/files2/jmyers/public_html/bugtraq/0166.html?30#mfs`.

[35] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proc. of the Symp. on Operating System Principles*, 1993.

[36] Christian Wettergren. Re: Mime question..., March 1995. Post to `bugtraq` mailing list. `http://www.eecs.nwu.edu/cgi-bin/mfs/files2/jmyers/public_html/bugtraq/1995a/0759.html?30#mfs`.

Note: CERT advisories are available on the Internet from
`ftp://info.cert.org/pub/cert_advisories/`.

8lgm advisories can be obtained from
`http://www.8lgm.org/advisories/`.

Figure 2: Sample configuration file

```
basic

putenv display
putenv HOME=. TMP=. PATH=/bin:/usr/bin:/usr/ucb:/usr/local/bin:/usr/local/X11/bin
:/usr/bin/X11:/usr/contrib/bin:/usr/local/bin XAUTHORITY=./.Xauthority LD_LIBRARY
_PATH=/usr/local/X11/lib

tcpconnect allow display

path super-deny read,write,exec */.forward */.rhosts */.klogin */.ktrust

# this is the paradigm to deny absolute paths and allow relative paths
# (of course, we will later allow selected absolute paths)
# assumes someone will put us in a safe sandboxed dir.
# note that the wildcard `*' can match anything, including /
path allow read,write *
path deny read,write /*

# allow certain explicit paths
path allow read /dev/zero /dev/null /etc/netconfig /etc/nsswitch.conf /etc/hosts
/etc/resolv.conf /etc/default/init /etc/TIMEZONE /etc/magic /etc/motd /etc/servic
es /etc/inet/services /etc/hosts /etc/inet/hosts

# note: subtle issues here.
# make sure tcpconnect is loaded, to restrict connects!
# /dev/ticotsord is the loopback equivalent of /dev/tcp.
path allow read,write /dev/tcp /dev/ticotsord

# where libraries live; includes app-defaults stuff too
path allow read /lib/* /usr/lib/* /usr/local/X11/lib/* /usr/local/X11R6/lib/* /us
r/share/lib/zoneinfo/* /usr/local/lib/* /usr/openwin/lib/*

# these are here so you can read the files placed by Netscape and Mosaic
path allow read /var/tmp/* /tmp/*

# this is where binaries live; it should look a lot like your PATH
path allow read,exec /bin/* /usr/bin/* /usr/ucb/* /usr/local/bin/* /usr/local/X11
/bin/* /usr/bin/X11/* /usr/contrib/bin/* /usr/local/bin/*
```

A DNS Filter and Switch for Packet-filtering Gateways

Bill Cheswick
Bell Laboratories
ches@bell-labs.com

Steven M. Bellovin
AT&T Research
smb@research.att.com

Abstract

IP-transparent firewalls require access to the external Domain Name System (DNS) from protected internal hosts. Misconfigurations and misuse of this system can create internal administrative and security problems.

Dnsproxy provides access to and protection from untrusted DNS services. It runs on a firewall, or on a trusted host just inside the firewall. The program receives (or intercepts) DNS queries and forwards them to an appropriate internal or external "realm" for processing. The responses can be checked, filtered, and modified before they are returned to the requester. The logging and consistency checks can provide information about possible DNS attacks and irregularities that are not available from most DNS implementations.

Introduction

For many years we have run application-level gateways on our interface to the Internet.[2][3, pp. 85–118] Behind these firewalls has grown an intimate community of perhaps 300,000 hosts, plus a (mostly) separate domain name system (DNS) arrangement. We've run our own DNS root and shielded ourselves from most of the security and administrative problems associated with the external DNS.

Recently we have installed a dynamic, or smart, packet filter. This newer technology allows us to have most of the security promised by an application-level approach, with much better performance.

A dynamic packet filter acts very much like traditional router, except that it keeps track of each TCP circuit, and permits bidirectional packet flow for each TCP circuit until the connection is terminated. Our users no longer need modified client software or proxies to access the net—they can get there themselves.

This gateway is transparent to permitted IP packets flows, which means we need something like standard DNS access to the Internet. This brings two problems:

1. Internally we rely on address-based authentication extensively for services like *rlogin*, *rsh*, *rcp*, and NFS. DNS is a vital part of these services, and it is easily subverted. Our crunchy outside now has a liquid center.

2. It's going to take a while to wean our companies away from our internal DNS root servers and our old gateways. During the transition period, internal exposure to external DNS information must be controlled and limited. We can't have external root NS entries knocking around inside, confusing innocent resolvers about where the answers lie.

The first problem is generic, and Internet-wide. Current, and especially older versions of *bind*, are much too trusting of the answers they receive.

The second problem is of our own making, but we expect that many sites are facing similar problems as their gateway technologies evolve.

What are we Afraid of?

There are many problems with the DNS protocol and its implementation. We don't intend to enumerate all the problems here—see references[1][5][6][7] for a detailed list.

Most of the known attacks are based on the fact that many popular Internet protocols rely on name-based authentication. If a client connects to a host using *rlogin*, the server does a reverse lookup of the client's IP address, and consults a table of trusted clients. If an attacker can subvert the DNS, this mechanism is broken. An attacker can force the target to make a normal query, and return additional irrelevant glue records to that query. These glue records are cached, and consulted on the subsequent query, giving the wrong answer.

We have seen DNS packet injectors and related tools in hackers' toolkits captured by law-enforcement folks.

It was clear to us that we could not allow an external name server to give us any information about an internal host or network. The DNS reply had to be unpacked, examined, censored and filtered, repacked, and forwarded to the recipient. This is an application-level gateway for external DNS services.

We also wanted to filter out all external NS records, to keep from polluting our internal DNS tree. In fact, we filter any record that is not explicitly necessary and common. There are a lot of miscellaneous and apparently harmless DNS records. We won't pass them until there is a need. As a performance optimization, we do pass back inside NS records; this allows inside resolvers to contact insider servers directly on subsequent queries.

We considered checking the address returned in the A record. Could an attacker harm us if he said his host was on one of our internal nets. We couldn't think of an attack, but one turned up in February involving Java (see CERT Advisory CA-96.05, March, 1996, and [4]). It was easy to install the check, because we had the right tool in place.

Dnsproxy

Dnsproxy is a DNS switch and filter. To clients it looks like a name server. We run it on two internal hosts; the resolvers inside point to these hosts. The next section discusses other deployment options. The program looks up the address in a configuration file and forwards the request to an appropriate "realm". Two realms are typically used: "inside" and "outside", though more may be configured. (We could set up separate `att.com` and `lucent.com` realms if we wished.) A realm is typically served by at least two name servers. *Dnsproxy* will forward the request to the least-busy server.

The realm's name server sends its reply to *dnsproxy*, which examines the response. The following checks are made:

- Is the record malformed in some way? If so, drop it.
- Does the query in the response match the query we sent out? If not, drop the whole thing.
- Did the response come from the expected IP address? If so, drop the response. This can obviously be spoofed, but the check is cheap. On the other hand, it can cause problems when talking to some multi-homed DNS servers, as the answers may appear to come from a different address for that machine.
- In each resource record, do all domain names refer to the relevant realm? If a query for `inside.com` would normally be directed to realm `inside`, an answer containing it will not be tolerated from another realm. This rule has some administrative implications, discussed below.
- Is there a filter rule for the responding realm to drop or modify a particular resource record? If so, apply it.

Deployment Choices

There are several different ways in which a DNS proxy can be deployed. First, it can simply be a server within a comparatively small lab. Each machine has its `resolv.conf` file pointing to the server; it in turn queries the outside or inside as needed. A variant points the `forwarders` option of Bind at `dnsproxy`, to allow for local caching of responses. We use both of these variations now. A second method would be to have the proxy act as the root server within a large organization. As per normal practice, all unresolved queries would be redirected to the root; it would take appropriate action. The third way is to install the proxy as a filter in some sort of dynamic packet filter. The packet filter could intercept outgoing DNS queries and kick them up to *dnsproxy* for processing.

Each of these three schemes has its advantages and disadvantages. The third scheme is likely the best. Insiders see the same image of the world as do outsiders. NS records need not be deleted, save for those that refer to the inside domain. And it doesn't matter where a host learned of a DNS server's address; the address will just work.

The problem is that this deployment mechanism is very dependent on the details of your firewall. Not all firewalls permit this sort of dynamic processing—simple packet filters do not—and those that do differ widely in their design.

The second scheme works well for large organizations; it has the disadvantage that it produces an inconsistent view of the world. There are then two sets of root servers, the outside's and the inside's. A laptop that lives in both worlds would need two different configuration files.

It would be least intrusive to use local `resolv.conf` files or `forwarders` entries in `named.boot` files is the least intrusive mechanism. Only the local machine is affected; it does not require organization-wide deployment. Furthermore, it works even with ordinary packet filters. The problem is that this approach doesn't scale; it may not be feasible to change all of the machines in a large organization. As such, it

```
realm
        inside  135.104.2.10,135.104.26.141      error
        outside 192.20.225.4,192.20.225.9        default

switch
        outside any     www-db.research.att.com
        outside any     www.research.att.com
        outside any     ampl.com
        outside any     dnstest.research.att.com
        inside  any     att.com
        inside  any     ncr.com
        inside  any     lucent.com
        inside  any     attgis.com
        inside  any     135.in-addr.arpa
        inside  ptr     127.in-addr.arpa
        inside  any     11.192.in-addr.arpa
        inside  any     19.192.in-addr.arpa
        inside  any     94.128.in-addr.arpa
        inside  any     127.192.in-addr.arpa
        inside  any     222.131.in-addr.arpa
        inside  any     243.132.in-addr.arpa
        inside  any     206.141.in-addr.arpa
        inside  any     25.149.in-addr.arpa
        inside  any     52.153.in-addr.arpa
...
        inside  any     87.153.in-addr.arpa
        outside any     *

filter  outside block   * NS *
        outside block   * A 135.104/16
        outside block   * A 135.180/16
        outside block   * A 127/8
```

Figure 1: A sample configuration file.

is best for experimental use or within relatively small groups. We have found that the benefits of the dynamic packet filter have given our users strong incentive to point their resolvers at the *dnsproxy* service.

We are currently using this latter choice, precisely to avoid impact on the rest of the company. At this stage, the code is experimental, and we are not prepared to deploy it widely. Ultimately, we will likely modify it so that it can be integrated with our dynamic packet filter.

It's not surprising that we have found it vital to duplicate the *dnsproxy* service. Since *all* name server queries come through it, its host provides an annoying single point of failure. Each realm has at least two servers, as well.

Performance

We've made no attempt to optimize the code in *dnsproxy* at this point: we are pleased enough that it works well. At present, the program is a CPU hog. We are processing 40,000 requests an hour on an NCR 3430 server with two 60MHz Pentiums. It uses about 10 CPU seconds per minute.

But the users see snappy response times. Typical round-trip times are about 10ms.

Scheduling

Dnsproxy supports multiple servers per realm. How should we use these? It would be nice to share the load, and if one server goes down, to rely on the others. How do we tell if a server is down? We could ping the server, but that would require additional code, and possibly a new hole in the firewall. Why not use our DNS traffic flow?

The problem is that a server may fail to respond because it hasn't obtained an answer, or because it is down. With a hefty load such as ours, we get plenty of indications that a server is working, but it is harder to tell if it isn't.

At first we tried a moving average of server response times. They all started with a high average, and quick responses brought the average down. If we gave up on a query, we included the timeout in our average. This worked well enough, and wasn't hard to implement. But we found a simpler way.

Resolvers are accustomed to timing out, and reissuing requests: typical retries arrive within three seconds. We can use this flexibility to waste an occasional request in the name of scheduling. To do this, we keep track of the number of outstanding requests to each server. We send the request to the server with the shortest queue. Even under heavy load, we usually get our answer before a new query comes in. Typically, one server handles all of the load until a single query gets stalled. Then it switches to the other. If one server goes down, the second starts handling the entire load within a couple of requests.

Other approaches

Paul Vixie[7] is working to add these and similar features to Bind. We hope he succeeds. For example, he is modifying the forwarding code to implement checks like ours, and others.

On the other hand, Bind is already a very large program; adding more security-critical functionality to it may not be a good idea. We're likely to continue running *dnsproxy* no matter how the base code changes.

Administration details

Dnsproxy normally runs as a detached daemon, and listens to queries on UDP port 53. (To access this port, it currently runs as *root*, which it shouldn't.) It forwards its queries from a predictable UDP port based on the realm, so it is easy to install appropriate filtering rules in a gateway. For example, in the following realms:

```
inside  135.104.70.9  error
outside 192.20.225.4  default
```

queries to 135.104.70.9 port 53 would be forwarded from port 54 in *dnsproxy*, and queries to 192.20.225.4 would come from UDP port 55. If a debugging version of *dnsproxy* is listening to port 9953, these ports would be 9954 and 9955 respectively.

Figure 1 shows a sample configuration file. The `realm` section describes the realm name and servers. It also can process erroneous requests or be the default (keywords `error` and `default`) if a query does not match any entry in the `switch` section.

The `switch` section is an ordered list of query types and the realms that process them. When a query arrives, *dnsproxy* runs through the list to find the first match, and dispatches the query to the given realm. This crude data structure eats much of our processing time, and should probably be improved.

The `filter` section is for responses, and is specified per realm. In this example we explicitly filter out all NS records, and any A records that refer to our internal addresses. Other filtering occurs as well, as described above. Our filter rules are fairly primitive. One should probably be able to match any specific type of resource record. We haven't needed this generality yet, so the code is currently fairly crude.

Dnsproxy generally logs to *syslog*. The usual *syslog* level controls the detail and severity of logging information. At the debugging level it produces a full dump of every query and response, producing a torrent of output. At a typical *syslog* logging level of *notice*, only records with unusual reason codes are displayed.

The routine logging can show a host of ills and configuration mistakes that might be normally missed. After we sifted through these, we raised the logging level. Error logs should be normally silent, so really unusual events won't get buried in a sea of mundane trivia. It has been a bit difficult to get the logging level right.

Dnsproxy does not handle tcp requests at present. This hasn't been a problem in our environment, but there are name servers that rely on this ability.

Availability

At present this code is not available outside of Lucent and AT&T.

References

[1] Bellovin, S. *Using the Domain Name System of System Break-ins.* Proceedings of the Fifth Usenix Unix Security Symposium, June 1995, pps. 199–208.

[2] Cheswick, W. R. *The Design of a Secure Internet Gateway.* Proceedings of the Usenix Summer '90 Conference.

[3] *Firewalls and Internet Security; Repelling the Wily Hacker.* Cheswick and Bellovin. Addison Wesley, 1994.

[4] Dean, D. and Wallach, D. *Security Flaws in the HotJava Web Browser.* Proceedings of the IEEE Symposium on Security and Privacy, May 1996.

[5] Mockapetris, P. *Domain names–concepts and facilities.* RFC 1034, Nov. 1987. Updated by RFC1101.

[6] Mockapetris, P. *Domain names–concepts and facilities.* RFC 1035, Nov. 1987. Updated by RFC1348.

[7] Vixie, Paul. *DNS and Bind Security Issues.* Proceedings of the Fifth Usenix Unix Security Symposium, June 1995, pps. 209–216.

Confining Root Programs with Domain and Type Enforcement (DTE)[1] [2]

Kenneth M. Walker
(ken@tis.com)

Lee Badger
(badger@tis.com)

Michael J. Petkac
(mjp@tis.com)

Daniel F. Sterne
(sterne@tis.com)

Karen A. Oostendorp
(kaos@tis.com)

David L. Sherman
(sherman@tis.com)

Trusted Information Systems, Inc.
3060 Washington Road
Glenwood, MD 21738
<http://www.tis.com/>

Abstract

The pervasive use of root privilege is a central problem for UNIX security because an attacker who subverts a single root program gains complete control over a computing system. Domain and Type Enforcement (DTE) is a strong, configurable operating system access control technology that can minimize the damage root programs can cause if subverted. DTE does this by preventing groups of root programs from accessing critical files in inappropriate access modes. This paper illustrates how a DTE-enhanced UNIX prototype, driven by simple, machine-interpretable DTE policies, can provide strong protection against specific classes of attacks by malicious programs that gain root privilege. We present a sequence of policy components that protect system binaries against Rootkit, a widely used hacker toolkit, and protect password, system log, user, and device special files against other root-based attacks. Tradeoffs among DTE policy complexity, scope of protection, and other factors are discussed.

1. Introduction

The pervasive use of root privilege is a central problem for UNIX security because an attacker who subverts a single root program gains complete control over a computing system, including enterprise data, operating system components, cryptographic keys used for secure communications, and privileged network connections to other local hosts. Unfortunately, many daemons and other programs that execute with root privilege are complex and riddled with security vulnerabilities. For these reasons, root programs are extremely attractive targets for attackers. UNIX system lore is replete with examples of root programs being tricked into misusing their privileges for a variety of malicious purposes [5,9]. By providing and relying on a single all-powerful privilege, UNIX systems "put all their eggs in one basket."

Finding and fixing vulnerabilities in root programs is important; but it is unwise to assume that all such vulnerabilities can be found and fixed before attackers can exploit them. Domain and Type Enforcement (DTE) is an operating system access control technology offering a fundamentally different approach to this problem [1,14]. We assume that any UNIX system will include one or more root programs containing exploitable vulnerabilities. We then attempt to minimize the damage these programs can cause if subverted. Configuring an appropriate DTE access control policy places root programs in restrictive execution environments that only allow access appropriate to each program's assigned responsibilities, thereby curtailing unnecessary access to security-critical files. In effect, this places the UNIX system eggs in separate baskets.

---

[1] Approved for Public Release - Distribution Unlimited (#96-S-2589)

[2] This research was supported by ARPA contract DABT63-92-C-0020.

Previous papers on DTE have focused on the goals, design, and features of our DTE-enhanced UNIX prototype and have illustrated these via hypothetical policy examples [1,14]. This paper illustrates how DTE mechanisms driven by simple DTE policies can provide strong protection against specific classes of root-based attacks on UNIX. In particular, we present a simple, experimentally validated policy that protects UNIX against Rootkit [19], an increasingly popular hacker toolkit that attempts to overwrite system binaries. Policy extensions are then presented to demonstrate how other specific UNIX vulnerabilities can be addressed. These enhancements protect system log files, password files, and several device special files (i.e., /dev/kmem). Finally, we provide a policy extension that allows users to run web browsers in a restricted domain. This domain restricts the damage that can be caused by a browser or helper applications when interpreting malicious web pages. These policy extensions are combined into a composite DTE policy presented in the appendix.

2. DTE Background

DTE is an access control technology derived from the earlier work of Bobert and Kain [4,15] that restricts process access according to a site-specific security policy. A DTE system associates a *domain* with each running process and a *type* with each object (e.g., file, packet). As a DTE UNIX system runs, a kernel-level DTE subsystem compares a process's domain with the type of any file or the domain of any process it attempts to access. The DTE subsystem denies the attempt if the requesting process's domain does not include a right to the requested access mode for that type. DTE restricts root as well as non-root processes and operates in addition to traditional UNIX protection bits. Suitably configured, DTE partitions a system according to the principle of *least privilege*, which grants each program only those access rights needed to perform its assigned function. This is a well-established technique for increasing both the security and reliability of computing systems.

A characteristic of DTE that distinguishes it from other access control schemes [2,3,4] is its use of a human-friendly high-level language for specifying security policies. The DTE Language (DTEL) provides four primary constructs for specifying policies:

- The **type** statement declares equivalence classes of data (e.g., personnel, manufacturing, corporate proprietary) that are treated differently by the policy.
- The **domain** statement defines the access modes a process running in that domain is permitted to use when accessing objects of specified types (e.g., read, write, or execute) or interacting with processes in other domains. A process running in domain A may *transition* to another domain B only by executing one of B's entry point programs (via exec()). A process may transition to another domain by explicit request only if its domain includes the *exec* mode of access to the target domain. Alternatively, a process may be automatically transitioned if its domain includes *auto* mode access to the target domain. The auto transition feature allows a policy to unilaterally partition families of existing programs into separate domains without requiring code modifications. A domain may also provide the right to send specified UNIX signals to processes in other domains.
- The **initial_domain** statement determines the domain in which the system's first process (usually /etc/init) starts.
- The **assign** statement binds types of data to specific files or directory hierarchies of files.

DTE and DTEL are described in greater detail in previous papers [1,14]; this paper covers them only as needed to explain the policy examples below. THE DTE prototype on which these examples were developed is based on BSD/OS™[3] UNIX.

3. Protecting System Binaries from Rootkit

Hacker toolkits are now widely available through many sources, including the Internet. These toolkits allow attackers to break into systems by exploiting security holes. Some toolkits also help attackers cover their tracks afterward. An increasingly popular toolkit is Rootkit [19]. Once an attacker has penetrated a system and obtained root permission, Rootkit builds a hidden backdoor into the system for future access. Installing Rootkit modifies the standard UNIX login program to recognize special login names, skip normal access checks for those names, and provide a hidden session with root privileges. Rootkit modifies several other

[3] BSD/OS is a trademark of Berkeley Software Design, Inc.

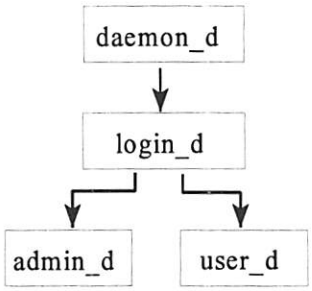


Figure 1. Domain Relationships

UNIX utilities, including ls, netstat, and ps, to hide the presence of the intruder after login.

3.1 Strategy

Our strategy for preventing installation of Rootkit is simple. Using DTE, we create a special administrative domain having write access to system binaries and their containing directories, and we allow transitions into that domain only after performing strong user authentication and authorization checks. All other processes, including root daemons, will run in less powerful domains that lack such write access. If an attacker subverts a root daemon, its accesses will be confined according to the daemon's domain, and it will be unable to replace login, ls, and other programs that constitute Rootkit's hidden backdoor.

The organization of a simple DTE policy that implements this strategy is depicted in Figure 1. The policy partitions all processes into four domains: 1) daemon_d, a domain for system daemons, including init; 2) login_d, a domain for the DTE-enhanced login program; 3) user_d, a domain for ordinary user sessions; and 4) admin_d, a domain for system administrator sessions. The daemon_d domain includes an access right for transitioning to the login_d domain so that the daemon that invokes login (getty) can start login in the proper domain. The login program acts as a gatekeeper, determining which user sessions should be started in the user_d and admin_d domains. Consequently, the login_d domain includes rights that allow transitions to those two domains. Although most system administration tasks will be carried out using root processes in the administrator domain, both root and non-root processes can exist in each of these four domains.

All information in files and other objects is categorized into five types: 1) generic information created by user processes (generic_t); 2) system binaries (binaries_t); 3) UNIX configuration files to which most processes need only read access (readable_t); 4) DTE security metadata (dte_t); and 5) miscellaneous system-generated information that many processes may need to update (writable_t).

3.2 Initial Policy Core

Figure 2 shows the core of a DTE policy that implements this strategy. Additional policy elements needed to loosen constraints on functionality or strengthen security are described later.

3.2.1 Daemon_d

The daemon_d domain is the domain in which the first system process, init, runs. This is indicated in the initial_domain statement and by the tuple showing that /sbin/init is the entry point for the daemon_d domain. All descendants of init will run in the daemon_d domain until one of them invokes /usr/bin/login, the entry point for the login_d domain. This causes an automatic transition of the login program into the login_d domain, as indicated by the tuple (auto->login_d). Note that invoking the login program is the *only* way for processes in this domain to transition to any other domain.

Like most domains, daemon_d's definition includes a comma-separated list of expressions of the form (modes->type). DTE modes include read (r), write (w), and execute (x). These are similar to UNIX modes except that execute does not include directory traversal, which is a separate mode (d). The default creation type (c), is an extension of write mode; it identifies the type attribute automatically associated with new objects if the creating process does not explicitly specify a type.

The tuple (rxd->binaries_t) allows a process in the daemon_d domain to read, execute, and search for system binaries but not modify them, even if the process has root privilege; this tuple implements a key aspect of our strategy to thwart Rootkit. In this domain, there is no type of file that is both executable and modifiable. As a result, processes in this domain cannot manufacture or import any executable that they can subsequently execute. Similarly, the tuple (rd->generic_t, readable_t) allows daemon processes to read but not modify various user-generated files and UNIX configuration files.

The assign statements at the bottom of Figure 2 assign the type readable_t by default to all files in /etc and the

type generic_t by default to all files in the file system that are not assigned a type by other assign statements. Additional assign statements can be added to label and protect additional configuration files (or any other file) in any part of the file hierarchy. Defaults for directories are indicated by the -r (recursive) flag. The -s (strict) flag indicates that the type associated with the pathname cannot be changed at runtime, even if the file is replaced by a different file. When combined with the -r flag, -s indicates that all files within the indicated directory will be labeled with the specified type; no other types will be permitted in the directory.

3.2.2 Login_d

The login_d domain includes the tuple (exec->user_d, admin_d). It is the only domain that can transition into the user_d or admin_d domains and is, hence, non-bypassable. Because the login_d domain is critical to our strategy, it has been designed so that only one binary can execute in it. That binary is the domain's entry point /usr/bin/login, the DTE-enhanced login program. Because this domain lacks execute (x) access to any type, any attempt by the login program to invoke any other binary without first transitioning to another domain (and shedding privilege) will be denied. This increases an attacker's difficulty of "commandeering" a process in this domain, should a means be discovered for penetrating the login program.

The DTE login program itself has been extended and strengthened in several ways. At login time, it obtains a role request from the user. If the user is authorized to assume that role, the login program invokes the user's shell (or other program) in the initial domain associated with that role, in this example, user_d or admin_d. The authentication and role authorization databases are labeled with DTE types (readable_t and dte_t), and access to these type is strictly controlled by the DTE

```
type generic_t, binaries_t, dte_t, readable_t, writable_t;

domain     daemon_d =(/sbin/init),
                     (crwd->writable_t),
                     (rxd->binaries_t),
                     (rd->generic_t, readable_t, dte_t),
                     (auto->login_d);

domain     login_d = (/usr/bin/login),
                     (crwd->writable_t),
                     (rd->generic_t, readable_t, dte_t),
                     setauth,
                     (exec->user_d, admin_d);

domain     user_d =  (/usr/bin/{sh, csh, tcsh}),
                     (crwxd->generic_t),
                     (rwd->writable_t),
                     (rxd->binaries_t),
                     (rd->readable_t, dte_t);

domain     admin_d = (/usr/bin/{sh, csh, tcsh}),
                     (crwxd->generic_t),
                     (rwxd->writable_t, binaries_t, readable_t,
                          dte_t),
                     (sigtstp->daemon_d);

initial_domain =     daemon_d;

assign     -r        generic_t        /;
assign     -r        writable_t       /usr/var, /dev, /tmp;
assign     -r        readable_t       /etc;
assign     -r -s     dte_t            /dte;
assign     -r -s     binaries_t       /sbin, /bin, /usr/libexec,
                                      /usr/{sbin,bin},
                                      /usr/local/bin;
```

Figure 2. DTE Policy Core for Protecting Binaries and Configuration Files

policy to protect these files from unauthorized modification. Furthermore, at session start, the login program establishes a separate, immutable process user ID, called the DTE UID, that cannot be modified via setuid programs or normal UNIX system calls. The setauth keyword in the login_d domain definition grants this domain the ability to set the DTE UID.

In addition, we are currently extending the DTE login program to require S/Key authentication if "SKey" is specified in the role authorization database [10]. In a later section, we illustrate how DTE can be used to protect authentication data, using password files as a familiar example. The techniques described, however, are equally applicable to other kinds of authentication data.

3.2.3 User_d

The user_d domain allows read and directory access to all types of information on the system, subject to the additional restrictions imposed by ordinary UNIX mechanisms. In addition, execute access is allowed for system binaries (binaries_t) and other files created in this domain (generic_t). The "c" in the tuple (crwxd->generic_t) indicates that, by default, any objects created in this domain will automatically be typed as generic_t. In addition, processes in this domain are permitted to modify existing objects of type writable_t or create new ones by explicit request. As indicated by the tuple (/usr/bin/{sh, csh, tcsh}), several common shells can be used as entry points into this domain. To non-administrative users running sessions in this domain, the system seems to behave like an ordinary UNIX system. Administrative users running in the user_d domain are prevented from carrying out certain administrative functions (e.g., rebooting the system, modifying system binaries), even after becoming the superuser. In order to carry out all administrative duties, administrative users must login to the admin_d domain via the login domain.

3.2.4 Admin_d

The admin_d domain allows read, write, execute, and directory access to all types of files on the system. This includes the capability to create and modify binaries, UNIX configuration files, and DTE policy files. This capability is not available in any other domain. DTE files are of type dte_t and are kept in the /dte directory, as indicated by the policy statement "assign -r -s dte_t /dte." In the admin_d domain definition, the "(sigtstp->daemon_d)" tuple indicates that processes running in the admin_d domain can send stop signals to processes in the daemon_d domain. This allows the admin_d domain to halt or reboot the system. No other cross-domain signals are permitted by this policy.

The policy core described above, together with a small amount of standard policy boilerplate, has been validated experimentally on a BSD/OS-based DTE prototype and shown to prevent installation of Rootkit while transparently allowing many normal system activities. We invoked the Rootkit installation script from a root shell in the daemon_d and user_d domains and verified that the script was unable to overwrite the login, ps, ls, and other binaries because of DTE access control checks.

3.3 Controlling Access to Password Files

In the initial policy example, it is not possible for regular users to change their system passwords or login shells. The password files on BSD/OS reside in /etc and therefore are of type readable_t. There are several ways this functionality deficiency can be handled. The simplest approach would be to provide the user_d domain with write access to readable_t. This approach is not acceptable because it allows any user who gains root privilege to change the password of any other user. This could allow an attacker in the regular user_d domain to change the password of a user authorized for the admin_d domain, thereby gaining access to a domain with the power to modify system binaries.

In order to better protect user authentication information from unauthorized modification, access to this information must be strictly controlled. To do this, the password files are assigned a new type (passwd_t) to which most domains are granted read-only access, as shown in the policy extension in Figure 3 and the appendix. The admin_d domain and a newly created domain (passwd_d) are permitted to write the password files.

Since regular UNIX authentication mechanisms are employed in this example, it might appear that the password programs (i.e., passwd, chpass, and chsh) should be used as entry points into the passwd_d domain. User_d's domain specification could force a domain transition whenever one of these programs is executed. This would ensure that only these programs are permitted access to the authorization information. However, this is not sufficient because it would allow root users to change other users' passwords. Since we assume that root access can be obtained illicitly, the UNIX UID cannot be trusted for user identification. The DTE UID, that is set only from the login_d

```
type  passwd_t;

domain user_d =    ...,
                   (auto->passwd_d),
                   ...;

domain passwd_d =  (/usr/bin/dtpasswd),
                   (crwd->passwd_t),
                   (rwd->readable_t, writable_t),
                   (rxd->binaries_t),
                   (rd->generic_t);

assign     -s      passwd_t     /etc/{master.passwd, spwd.db, pwd.db,
                                 passwd};
```

Figure 3. DTE Policy Extension for Passwd

domain, should be used instead, but the password programs would need to be modified to do this.

A less disruptive approach that allows the password programs to remain unmodified involves creating a simple wrapper (dtpasswd) for the programs. The wrapper program checks the requesting process's DTE UID to ensure that the user is not attempting to change someone else's password information and then executes the requested password program. The wrapper program is designated as the sole entry point to the passwd_d domain. This new domain is set up as an auto-transition from the user_d domain. If the user attempts to run a password program directly without using the wrapper, the program will run in the user_d domain and will not be allowed to update the password files. However, from the admin_d domain, password programs can execute unconstrained since the admin_d domain has write access to passwd_t. This allows the administrator to bypass the wrapper and change other users' passwords.

3.4 Providing Additional Assurance

The above policy protects a system's executables and configuration files from direct manipulation by attacks like Rootkit. Figure 4 and the appendix present an extension that protects these resources from other, more sophisticated attacks by users who obtain root privilege. One such class of attacks attempts to manipulate raw devices via device special files, bypassing normal access paths and their associated DTE constraints. Two types of device special files present particularly inviting targets. The disk device special files allow access to the disk, bypassing the file system controls. The memory device files (/dev/kmem, /dev/mem) allow direct access to system memory. Using these devices, an attacker could change data on disk or in memory and undermine the security policy.

```
type        mem_t, disk_t;

domain      daemon_d =  ...,
                        (auto->fsck_d),
                        ...;
domain      fsck_d =    (/sbin/{fsck, mount_mfs}),
                        (crwd->disk_t),
                        (rwd->writable_t),
                        (rd->generic_t, readable_t);

assign    -s      mem_t       /dev/{kmem, mem, core};
assign    -s      disk_t      /dev/{rsd0a, rsd0b, rsd0g, rsd0h,
                                  sd0a, sd0b, sd0h, sd0g};

mount             (/dev/sd0a,       /);
mount             (/dev/sd0h,       /usr);
mount             (/dev/sd0g,       /usr/home);
```

Figure 4. DTE Policy Extension for Device Special Files

To protect the memory device files, a new type is defined (mem_t), and access to this type is restricted. The memory device files do not need to be written by any process, though they are read by several of the daemons and user processes (e.g., ps, vmstat). Similarly, the disk device files can be protected by creating and assigning to them a new type (disk_t). Access to this type can be limited to a few administrative processes (i.e., fsck, newfs, mount_mfs) by placing these processes in a new domain (fsck_d). This eliminates the need for the daemon_d domain to retain any access to the disk devices.

Another possible attack is to create a new device special file that aliases a critical resource, such as the boot disk partition, and then freely manipulate that resource via the new device special file. The DTE prototype prevents these attacks by requiring that all device special files naming the same physical device have the same type and by allowing mknod commands only if the requesting process has read and write access to the type of the device. These mechanisms ensure that only processes in domains authorized by the DTE policy can manipulate devices. Note that this kind of protection can be extended to other kinds of devices (e.g., the console).

File system mount operations provide another potential means of attack. Because the DTE kernel binds type attributes implicitly to files based on directory hierarchies, an attacker might attempt to alter the hierarchies, thereby changing the type bindings in a way that lessens DTE constraints. For example, if the root user unmounts a file system and mounts it at a mount point that is recursively typed "foo_t", all of the files on the file system would have their types changed to foo_t. To prevent this, DTEL provides "mount constraints" that restrict where file systems can be mounted. The DTEL mount constraints contain information similar to that in a UNIX fstab file. In addition to ensuring that mount operations conform to the constraints, the DTE prototype also prevents modification (e.g., rename) of device special files that are mount constrained. These mechanisms ensure that rogue root programs cannot disturb the type associations but allow for flexibility for root to remount disk partitions in cases where remounting would not affect type bindings.

Protecting the device special files and mount operations, addresses a significant backdoor through the security of most UNIX systems.

4. Broader Protection

The previous section describes how a simple DTE policy can protect system binaries from Rootkit and other root-based attacks. This section illustrates how small policy additions can further extend DTE protection to other important information resources, in particular, system logs, administrative functions, and user data.

4.1 Protecting System Logs

Attackers often attempt to "cover their tracks" by modifying or disabling the attacked host's audit system. This threat can be countered by another simple policy extension as shown in Figure 5 and the appendix. First, the audit log files are given their own type, syslog_t. Second, a new domain (syslog_d) is created for the syslogd daemon. Only this domain and the admin_d domain are permitted to write the syslog_t type. All other domains are given either no access or read-only access to the syslog_t type. The admin_d domain needs write access to this type to allow for controlled cleanup of the log files. Daemon_d will be permitted (in fact, required) to auto-transition into syslog_d when

```
type  syslog_t;

domain      daemon_d =  ...,
                        (auto->syslog_d),
                        ...;
domain      syslog_d =  (/usr/sbin/syslogd),
                        (crwd->syslog_t),
                        (rwd->writable_t),
                        (rd->readable_t, generic_t);

assign -r   syslog_t    /usr/var/{log, run/syslog.pid};
assign      writable_t  /usr/var/log/{wtmp, lastlog, utmp, lpd-errs};
```

Figure 5. DTE Policy Extension for Syslogd

the syslogd program is executed. Although not demonstrated here, other log files (e.g., utmp, wtmp, lpd log) can also be protected using these techniques.

4.2 Providing Multiple Administrative Roles

In many environments, it is prudent to split administrative duties into several parts so that some administrative duties can be performed by individuals whose privileges are limited. As an example of how this can be accomplished using DTE, in Figure 6 and the appendix, we have derived from the admin_d domain a limited UNIX administrative domain. To more clearly distinguish between these two domains, we have renamed admin_d to dte_admin_d. These domains are designated as initial domains for the UNIX administrator and the DTE administrator roles. Users operating in the UNIX administrator role will be allowed to modify system binaries and configuration files but not the DTE configuration or password files. DTE administrators will be permitted to perform all administrative duties. Access to either of these roles is controlled by login and can be constrained with appropriate strong authentication. The UNIX administrator is not permitted to change other users' passwords because this would allow a UNIX administrator to change a DTE administrator's password and assume the DTE administrator role. Although the UNIX administrator can prevent the system from operating at all by, for example, removing the password file, the UNIX administrator cannot alter the password file contents because of the -s flag in Figure 3's assign statement.

4.3 Securing a Web Browser

The World Wide Web was originally used as an easy way to share documents. Gradually, full-featured languages, like Java, have been developed that extend this technology. Using Java, a browser can retrieve code (an applet) from a remote machine and interpret or execute the code locally. Conceptually, this technology provides a good way to remove processing bottlenecks; however, its security implications are considerable. It provides an enormous opportunity to exploit a new genre of attacks in which code is retrieved from unknown, distant sites, at a user's request, or possibly without the user's knowledge, and executed locally with the user's full access privileges. Although the Java environment purports to prevent hostile applets from harming a browser's host and surroundings, a number of critical security flaws in the Java environment have already been discovered [6]; we expect that additional flaws will be identified in the future. Additionally, the Web is routinely used to download other kinds of executable content, such as, TCL and PostScript. Generally, web browsers do not run with root privilege; however, a browser-based attack that gains root privilege by exploiting a hole in a setuid program is clearly possible. Moreover, even without root, an unconstrained browser could inflict significant damage on user data.

Using DTE, users can run web browsers in a constrained domain that restricts access to critical user files. Figure 7 and the appendix show a policy extension that constrains the write accesses of NCSC's Mosaic web browser and Netscape's web browser. In addition to files of type writable_t, the only files that these browsers need to write are: 1) the "hotlist" and other files that the browser must update to capture

```
domain      login_d      =      ...,
                                (exec->unix_admin_d, dte_admin_d),
                                ...;
domain      unix_admin_d =      (/bin/{sh,csh,tcsh}),
                                (crwxd->generic_t),
                                (rwxd->binaries_t, writable_t, readable_t,
                                      syslog_t, disk_t, mem_t),
                                (rd->dte_t, passwd_t),
                                (sigtstp->daemon_d),
                                (auto->passwd_d);
domain      dte_admin_d =       unix_admin_d,
                                (rwxd->dte_t, passwd_t);
```

Figure 6. DTE Policy Extension for Multiple Administrative Roles

```
type        browser_t;

domain      user_d =          ...,
                              (auto->browser_d),
                              ...;
domain      browser_d =       (/usr/X11R6/bin/{Mosaic, netscape}),
                              (crwd->browser_t),
                              (rwd->writable_t),
                              (rxd->binaries_t),
                              (rd->generic_t, readable_t, passwd_t);

assign    -r      browser_t   /usr/home/ken/{.MCOM-HTTP-cookie-file,
                              .MCOM-preferences, .MCOM-global-history,
                              .MCOM-cache, .MCOM-bookmarks.html};
assign    -r      browser_t   /usr/home/ken/{.mosaic-global-history,
                              .mosaic-hotlist-default, .mosiacpid,
                              .mosaic-personal-annotations);
assign -s -r      browser_t   /usr/home/ken/scratchpad;
```

Figure 7: DTE Policy Extension for a World Wide Web Browser

history between sessions; and 2) other files in a designated scratchpad subdirectory in the user's home directory, in this case, /usr/home/ken/scratchpad. The browser may download files into this subdirectory for printing or other purposes. These files and the scratchpad are assigned the type browser_t, the only other type writable in the browser's domain browser_d. As a result, in this domain, the browser cannot delete, modify, or overwrite any other user data file on the system, even if root privilege is obtained. This is because user data files and directories are of type generic_t, a type not writable in the browser's domain.

In addition, all files created in the browser's domain are labeled with a type (browser_t) to which neither the browser, its progeny, nor any other process running in an ordinary user's domain has execute access rights. This means that the browser cannot import or manufacture binaries that it or other processes in the user_d or daemon_d domains can execute. As a consequence, the browser cannot plant trojan horse binaries that ordinary users might execute inadvertently. An auxiliary domain could also be provided that would allow users, but not browsers, to regrade downloaded files to another type that is executable, e.g., generic_t. This would require the user to explicitly regrade files before being able to execute them.

Note that the assign statements in Figure 7 are specific to the user ken. Similar assign statements are needed for each user that runs browsers in the browser_d domain. These statements can be automatically generated by a tool whenever a new user account is established.

5. Discussion

Figure 8 depicts the domain transition relationships represented by the combination of policy components described in sections 3 and 4. Although these relationships are more numerous than those depicted in Figure 1, they are nevertheless easy to understand and extend further if needed.

5.1 DTE Policies For Operational Systems

The policy components above have undergone limited testing to verify that they protect against the kinds of root-based attacks we cite. In addition, we have verified that they allow a significant range of common UNIX programs to operate normally, including vi, emacs, ps, rlogin, telnet, make, gcc, and mail. We are currently using the policy presented in the appendix on our software development workstations.

Operational use of DTE systems by people who are not DTE developers, however, will likely require further policy extensions and DTE-aware versions of a few standard utilities. For example, a DTE-aware cron daemon may be needed that can start system and user programs in the domain of a requester or any domain into which a requester is authorized to transition. This requires that cron's own domain include the ability to transition into domains of requesters. Another example is preauthenticated rlogin. Providing security for preauthenticated rlogin in the presence of possible root penetrations requires that all inputs into the login_d domain be conveyed over DTE-protected paths; these inputs include file descriptors, command-line arguments, and DTE UID's received from the network.

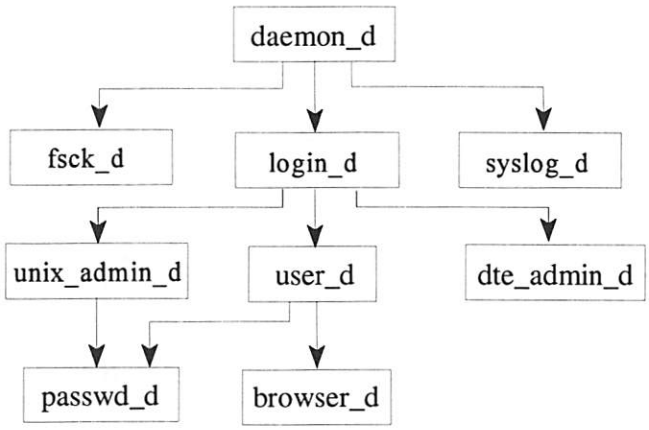


Figure 8. Final Domain Relationships

Protecting these input paths requires, in turn, that getty, rlogin, and other trusted ancestors of login be segregated from other processes and placed into their own domain[s].

5.2 Security Tradeoffs

Perfect security is an impractical goal. Effective security, on the other hand, requires that security measures be deployed selectively and in a manner that balances security against other competing concerns. One of our research objectives is to build strong, flexible mechanisms that enable organizations to make appropriate tradeoffs involving operating system security.

One tradeoff illustrated above is the tradeoff between extent of protection and policy simplicity. Finer-grained control over the behavior of root processes can be gained by partitioning a system into more types and domains, where each domain provides minimal essential access rights for a group of related processes. Increased partitioning, however, results in a DTE policy that is larger, more complex, and less easily understood and maintained. This phenomenon is illustrated by the progression of incremental policy extensions above. The initial policy core is simple but protects against a limited class of threats; the composite policy provided in the appendix is more complex but provides significantly broader protection.

Another tradeoff is between policy simplicity and the use of unmodified programs. The examples above attempt to minimize the number of standard UNIX programs that must be modified because of DTE. Modifying more programs enables equivalent policies to be made simpler. Consider the domain for the syslog daemon described in section 4.1. Syslog needs to update the system log, which is of type syslog_t. Syslog also must create a UNIX domain socket for receiving logging requests from clients. Since the existing syslog program is not DTE-aware, the file system pathname that is created for this socket will also be of type syslog_t. Clients must open this socket (and pathname) to communicate with syslog. Consequently, the right to read the type syslog_t must be added to the domains of syslog clients, making each of them somewhat more complex. The need to augment these domains can be avoided by modifying syslog so that it creates this socket with type writable_t.

5.3 Comparison With Conventional Techniques

Chroot is the principle UNIX mechanism for placing root processes in restrictive environments. Chroot, however, can be cumbersome to use and also can usually be subverted by root processes.

- Each chroot'd environment must contain its own copy of each file needed by the process(es) that will run in that environment. Setting up numerous chroot'd environments is therefore inconvenient and can waste disk space. Moreover, when environment updates are necessary, the presence of multiple file copies makes maintaining consistency across environments more difficult.

- Standard system services such as ptrace, signals, swapon, and mknod are available to the root user for taking control of non-chroot'd processes or the system itself.

In contrast, DTE permits files to be made selectively inaccessible to different processes without file copying. More important, DTE can provide stronger protection than chroot'd environments because its underlying mechanisms provide no special exemptions for root programs. The primary drawback of DTE as compared with chroot is that DTE requires a modified kernel.

Tripwire and COPS are security configuration checking tools that protect system binaries and other critical files but in a more limited manner than DTE. When run initially, Tripwire [12] computes and stores cryptographic checksums for system binaries and other security-critical files. When run subsequently, it recomputes these checksums and compares them with the stored values. If a binary has been replaced or modified, the checksums will not match. When Tripwire detects a mismatch, it notifies the security administrator.

COPS [8] compares ownership, permission bits, and contents of security-relevant files to sets of security expectations derived from established administrative practices. For example, it verifies that /bin and /etc are not world writable and that device special files are not world readable. It also detects poorly chosen passwords. COPS produces reports that point out potential insecurities in the system's current configuration.

In terms of protecting system binaries and other critical files, the most important difference among the DTE prototype, COPS, and Tripwire is that the first can *prevent* malicious modification while the other two cannot. Instead, Tripwire detects such modifications after the fact. COPS simply warns about critical files that are unnecessarily exposed to attack by unprivileged programs. If an attacker obtains root privilege, COPS provides little help and can be disabled.

COPS and Tripwire have been ported to a variety of UNIX systems; neither requires kernel modifications.

6. Related Work

The designers of a number of UNIX-oriented secure operating systems have attempted to eliminate root privilege or reduce its potential for misuse. Sidewinder™[4] [16], an Internet firewall, is based on a version of BSD/OS UNIX that has been extended to include type enforcement, which it uses to impose additional constraints on root and non-root processes. Sidewinder includes two different kernels. Under the operational kernel, all processes are governed by type enforcement [4] while "security policy checks are bypassed" [16] in the administrative kernel, which is used to install software. The administrative kernel can only be entered by a user who is "physically connected to the Sidewinder" [16] and only after shutting down the operational kernel. Sidewinder includes a fixed, vendor-supplied access control configuration not intended to be modified by customers. This is entirely appropriate since Sidewinder is an embedded turnkey system (i.e., a fixed-function device); moreover, it increases the likelihood that the access controls have been and will remain configured properly.

By contrast, the DTE prototype is intended to be a general purpose UNIX system whose security mechanisms are easily configured by user organizations in accordance with their own security objectives, policies, and tradeoff decisions. This paper explains how these mechanisms can be configured effectively. The DTE prototype provides a single kernel for both operational and administrative purposes. This allows an organization to use DTE to define and delimit the rights of administrative personnel as appropriate to the organization's needs. It also allows administrative tasks, including installing software and extending a system's DTE policy, to occur without rebooting.[5] Moreover, administrative tasks for multiple user workstations can be carried out from a single administrator workstation via networking.

Trusted Mach®[6] [17], Trusted XENIX™[7] [18], and HP-UX CMW [13, 11] use other strategies to mitigate root-related vulnerabilities. Each has been designed to protect classified information from leakage in

---

[4] Sidewinder is a trademark of Secure Computing Corporation

[5] Features to restrict "on-the-fly" policy changes are currently under development.

[6] Trusted Mach is a registered trademark of Trusted Information Systems, Inc.

[7] XENIX is a trademark of Microsoft Corporation.

accordance with the Trusted Computer Systems Evaluation Criteria [7]. Trusted Mach effectively runs a separate UNIX operating system (OS) at each security classification, e.g., unclassified or secret. Each user process also has a classification and can communicate only with the corresponding OS. Strong separation between the OSs and clients at different classifications is enforced by a trusted computing base completely independent of UNIX mechanisms. As a result, an adversary that obtains UNIX root privilege at one classification gains no additional access rights to information at other classifications; hence security, in the sense of preventing leakage, is preserved.

Trusted XENIX is a modified UNIX operating system that supports user processes at multiple security levels concurrently. In Trusted XENIX, root privilege has been replaced by a collection of 36 specific privileges that selectively allow use of privileged system calls and non-privileged system calls with privileged options. For example, a call to audit() succeeds only if the caller possesses the AUDIT privilege. Privileges are encoded in a bit vector associated with each executable file and stored in an extended inode structure. When a process executes a program, the process obtains the program's privileges. Trusted XENIX defines five hierarchical roles for administrative users.

Like Trusted XENIX, HP-UX CMW supports user processes at multiple security levels concurrently. Similarly, HP-UX replaces root privilege with a collection of finer-grained privileges, each of which grants a process the right to invoke a particular action such as setting the system clock. HP-UX CMW provides three predefined administrative roles.

A fundamental difference between the DTE prototype and both Trusted XENIX and HP-UX is that the latter two have a predefined privilege structure that is hard-coded into their kernels. By contrast, the DTE prototype relies only minimally on the notion of predefined privileges. Instead, it allows user organizations to choose the kind and granularity of protection appropriate to their needs, recognizing that these needs may evolve over time and that unnecessary granularity may increase the cost and difficulty of administering and maintaining a system.

7. Conclusion

The pervasive use of the all-powerful root privilege is one of the most important sources of security problems in UNIX systems. This paper has presented a DTE policy that thwarts Rootkit and other root-based attacks that attempt to modify or replace critical system binaries, e.g., login. This policy has been experimentally validated on our BSD/OS-based DTE prototype as preventing Rootkit from installing malicious binaries while allowing a variety of unmodified UNIX programs to be used normally. In addition, we have presented a sequence of small, incremental policy extensions that provide broader protection against other root-based attacks. While these extensions do not address all known attacks, they do provide strong protection for several critical UNIX abstractions, including password files, raw disk devices, kernel memory, mount operations, and system audit logs.

The DTE prototype is intended to be a general purpose system whose security mechanisms can be configured by user organizations in accordance with their own security concerns and tradeoffs. The policy examples presented here illustrate that very simple DTE policies can provide limited but useful protection beyond that provided by ordinary UNIX. Moreover, they illustrate that broader protection can be provided by making policies more complex. The policies presented here are predicated on minimizing the number of existing UNIX utilities that must be modified. Additional policy simplifications are possible in some cases, if minor modifications are made to particular UNIX programs.

The results presented herein represent work in progress. We are now beginning to use the above policies to protect the UNIX workstations we depend on for routine, daily computing. We expect to continue improving the security of these workstations against root-based attacks by refining and extending these policies and the DTE prototype.

Bibliography

1. L. Badger, D. F. Sterne, D. L. Sherman, and K. M. Walker, "A Domain and Type Enforcement UNIX Prototype," USENIX Computing Systems, Vol 9, No.1, Winter 1996, pages 47-83.

2. D.E. Bell and L. LaPadula, "Secure Computer System: Unified Exposition and Multics Interpretation," Technical Report No. ESD-TR-75-306, Electronics Systems Division, AFSC, Hanscom AF Base, Bedford, MA, 1976.

3. K.J. Biba, "Integrity Considerations for Secure Computer Systems," USAF Electronic Systems Division, Bedford, MA, ESD-TR-76-372, 1977.

4. W.E. Boebert and R.Y. Kain, "A Practical Alternative to Hierarchical Integrity Policies," Proceedings of the 8th National Computer Security Conference, Gaithersburg, MD, p. 18, 1985.

5. W. Cheswick, and S. Bellovin, Firewalls and Internet Security: Repelling the Wily Hacker, Addison-Wesley, 1994.

6. D. Dean, E.W. Felten, and D.S. Wallace, "Java Security: From HotJava to Netscape and Beyond," Proc. of IEEE Syposium on Security and Privacy, Oakland, CA, May 6-8.

7. *Department of Defense Trusted Computer System Evaluation Criteria*, DoD 5200.28-STD, December 1985.

8. D. Farmer, "The COPS Security Checker System," Proceedings of the Summer 1990 USENIX Conference, Anaheim, CA, p. 165.

9. S. Garfinkel, G. Spafford, Practical UNIX Security, O'Reilly and Associates, 1991.

10. N. Haller, "The S/Key One-Time Password System," Proc. Internet Society Symposium on Network and Distributed System Security, San Diego, CA, February, 1994.

11. "HP-UX CMW Compartmented Mode Workstation Version 10.9," <http://www.dmo.hp.com/Fed/tac4/CMW.10.6.html>, Hewlett-Packard Company, May 15, 1996.

12. G. Kim, E. Spafford, "The Design and Implementation of Tripwire: A File System Integrity Checker," Purdue Technical Report CSD-TR-93-071, November 1993.

13. "Secure Web Platform Whitepaper," http://www.sware.com/papers/, SecureWare Inc., February 2, 1996.

14. D.L. Sherman, D. F. Sterne, L. Badger, S.L. Murphy, K.M. Walker, and S.A. Haghighat, "Controlling Network Communication With Domain and Type Enforcement," Proc. 18th National Computer Security Conference, pages 211-220, Baltimore, MD, 1995.

15. D. J. Thomsen, "Role-based Application Design and Enforcement," in Proc. of the Fourth IFIP Workshop on Database Security, Halifax, England, September 1990.

16. D. J. Thomsen, "Sidewinder: Combining Type Enforcement and UNIX," Proc. 11th Computer Security Applications Conference, Orlando, FL, December 1995.

17. "Trusted Mach Philosophy of Protection", Edoc-0003-93B, Trusted Information Systems, Inc., May 1993.

18. *Trusted XENIX Version 3.0 Final Evaluation Report*, CSC-EPL-92/001, National Computer Security Center, Fort Meade, MD, April 8, 1992.

19. W. Venema, "Root Kit," Presentation at SURFnet CERT-NL SGG-SEC/SSC Workshop, May 1995.

Appendix

This appendix presents the complete policy developed in this paper. In addition to the elements discussed above, this policy contains statements that address two network issues: DTE labeling of protocol control messages that carry no user data, and DTE mediation of interactions with remote systems that are not running DTE.

1) In order to appropriately label TCP protocol control messages that do not contain user data (and are not explicitly sent by user processes), the DTE prototype uses a standard domain (tcp_d) and type (tcp_t); these domain and type labels identify the TCP subsystem as the producer of protocol control information and are needed by DTE policies for systems employing TCP.

2) In order to mediate interactions with non-DTE hosts, the DTE prototype associates a domain with each such host and mediates interactions with processes running on that host as though all of its processes were running in the associated domain. The policy presented here uses the "inet_assign" DTEL statement to associate the "non_dte_d" domain with all non-DTE hosts (represented by the wildcard 0.0.0.0 IP address), thereby restricting interactions with such hosts to types specified in non_dte_d.

```
/*
 *    File: /dte/policy/dt_policy
 *
 *    DTE policy file.
 */

/*******************   Type Section   **********************************/

type
      binaries_t, // System executables
      dte_t,      // DTE configuration files
      generic_t,  // User data
      readable_t, // System configuration files
      writable_t, // System-created data
      syslog_t,   // Log files
      mem_t,      // Kernel memory device special files
      disk_t,     // Disk device special files
      passwd_t,   // Files used for user authentication
      browser_t,  // Files written by a web browser
      tcp_t;      // System generated TCP overhead data
                  // only used when no user data is sent

/*******************   Domain Definition Section   **********************/

domain      tcp_d  =     (crw->tcp_t);
domain      non_dte_d =  (crw->generic_t),
                         (r->binaries_t, dte_t, readable_t, writable_t,
                               tcp_t, browser_t);

domain      daemon_d =   (/sbin/init),
                         (crwd->writable_t),
                         (rxd->binaries_t),
                         (rd->generic_t, readable_t, dte_t, syslog_t, mem_t,
                               passwd_t),
                         (r->disk_t),
                         (auto->login_d, syslog_d, fsck_d);

domain      fsck_d =     (/sbin/{fsck, mount_mfs}),
                         (crwd->disk_t),
                         (rwd->writable_t),
                         (rd->generic_t, readable_t);

domain      syslog_d =   (/usr/sbin/syslogd),
                         (crwd->syslog_t),
                         (rwd->writable_t),
                         (rd->readable_t, generic_t);

domain      login_d =    (/usr/bin/login),
                         (crwd->writable_t),
                         (rd->generic_t, readable_t, dte_t, syslog_d,
                               passwd_t),
                         setauth,
                         (exec->user_d, dte_admin_d, unix_admin_d);

domain      user_d =     (/bin/{sh,csh,tcsh}),
                         (crwxd->generic_t),
                         (rwd->writable_t),
                         (rxd->binaries_t),
                         (rd->syslog_t, readable_t, dte_t, mem_t,
                               passwd_t, browser_t),
                         (auto->passwd_d, browser_d);
```

```
domain      passwd_d =  (/usr/bin/dtpasswd),
                        (crwd->passwd_t),
                        (rwd->writable_t, readable_t),
                        (rxd->binaries_t),
                        (rd->generic_t);

domain      browser_d = (/usr/X11R6/bin/{Mosaic, netscape}),
                        (crwd->browser_t),
                        (rwd->writable_t),
                        (rxd->binaries_t),
                        (rd->generic_t, readable_t, passwd_t);

domain      unix_admin_d =    (/bin/{sh,csh,tcsh}),
                        (crwxd->generic_t),
                        (rwxd->writable_t, binaries_t, readable_t, syslog_t),
                        (rwxd->disk_t, mem_t),
                        (rd->dte_t, passwd_t),
                        (auto->passwd_d),
                        (sigtstp->daemon_d);    /* System reboot */

domain      dte_admin_d =     unix_admin_d,
                        (rwdx->dte_t, passwd_t);

initial_domain =  daemon_d;

mount           (/dev/sd0a,     /);
mount           (/dev/sd0h,     /usr);
mount           (/dev/sd0g,     /usr/home);

inet_assign non_dte_d   0.0.0.0;

/*******************   Attribute Association Section   *********************/

// Default type for all files.
assign -r         generic_t   /;

// Protect security information
assign -r -s      dte_t       /dte;

// Types for executable areas
assign -r         binaries_t  /bin, /sbin, /usr/{bin, sbin};
assign -r         binaries_t  /usr/contrib/bin, /usr/libexec;
assign -r         binaries_t  /usr/local/bin, /usr/local/etc;
assign            binaries_t  /etc/uucp/{daily, weekly, uuxqt_hook};

// Writable areas
assign -r         writable_t  /usr/var, /dev, /tmp;
assign            writable_t  /dte/dt_diag;
assign            writable_t  /usr/var/log/{wtmp, lpd-errs};

// User log areas
assign            writable_t  /usr/var/log/{lastlog, utmp};

// Read-only areas
assign -r         readable_t  /etc;

// System log areas
assign -r         syslog_t    /usr/var/{log, run/syslog.pid};
```

```
// Password files
assign    -s      passwd_t    /etc/{master.passwd, spwd.db, pwd.db,
                                  passwd};

// Critical device special files
assign      -s    mem_t /dev/{kmem, mem, drum};

assign     -s     disk_t      /dev/{rsd0a, rsd0b, rsd0g, rsd0h,
                                  sd0a, sd0b, sd0g, sd0h};

// Browser writable files
assign -r         browser_t   /usr/home/ken/{.MCOM-HTTP-cookie-file,
                              .MCOM-preferences, .MCOM-bookmarks.html,
                              .MCOM-global-history, .MCOM-cache};
assign -r         browser_t   /usr/home/ken/{.mosaic-global-history,
                              .mosaic-hotlist-default, .mosaicpid,
                              .mosaic-personal-annotations};
assign -r -s      browser_t   /usr/home/ken/scratchpad;
```

SSH – Secure Login Connections over the Internet

Tatu Ylönen <ylo@ssh.fi>
SSH Communications Security Ltd.
Tekniikantie 12, FIN-02150 ESPOO, Finland
Tel. (intl) +358-0-4354 3205 fax +358-0-4354 3206

June 7, 1996

Abstract

SSH provides secure login, file transfer, X11, and TCP/IP connections over an untrusted network. It uses cryptographic authentication, automatic session encryption, and integrity protection for transferred data. RSA is used for key exchange and authentication, and symmetric algorithms (e.g., IDEA or three-key triple-DES) for encrypting transferred data.

SSH is intended as a replacement for the existing `rsh`, `rlogin`, `rcp`, `rdist`, and `telnet` protocols. SSH is currently (March 1996) being used at thousands of sites in at least 50 countries. Its users include top universities, research laboratories, many major corporations, and numerous smaller companies and individuals.

The SSH protocol can also be used as a generic transport layer encryption mechanism, providing both host authentication and user authentication, together with privacy and integrity protection.

1 Introduction

The Internet has become the most economical means for communication between two remote sites. Its uses include communicating with clients, connecting remote offices, file transfer, remote systems administration, banking services, working at home, and many others.

However, the Internet does not provide any protection for the transmitted information, and can become an information security nightmare for companies connected to it. Firewalls and access controls such as one-time passwords do not fully solve the problem, as it is easy to record and analyze any transmitted data or to hijack an already established connection and use it to attack machines inside the firewall.

The threats from the Internet include:

- Network monitoring: it is easy to record passwords, financial data, private messages, or corporate secrets from the network.
- Connection hijacking: it is possible to hijack a connection without either party noticing it, insert new commands at the command prompt, and remove the output of those commands from the output sent to the user. The same mechanisms can, for example, be used to manipulate remote banking connections to make wire transfers with a different sum and account than what the user thinks (and sees). Having the accounts protected by one-time passwords does not help.
- Routing spoofing: standard routing protocols and commonly used router configurations permit anyone in the world to reconfigure routings. This can be used to bring connections into networks through which they do not normally go, and where they can be hijacked.
- DNS (domain name server spoofing): active network-level attacks can be used to make name servers return whatever data is beneficial for further attacks. Verification by reverse-mapping does not help. The same holds for practically all widely used network services.
- Denial of service attacks: here the purpose is to prevent others from using a particular service. The simplest implementation of this attack is to overload the target machine with requests; however, more subtle forms are available, such as reconfiguring the routers so that packets no longer go to the machine, hijacking connections to the machine and returning erroneous results, etc.

The current IP protocol does not in any way guarantee any aspects of information security (e.g., authentication, privacy, data integrity). As higher level protocols are mostly based on the assumption that the lower level protocol can be trusted – and as this

is not the case – the higher level protocols aren't any more reliable. If security is needed, it must be entirely implemented on the application level.

An acceptable solution must guarantee at the same time authentication of both ends of the connection, secrecy of transmitted information, and integrity of transmitted data. For example, if only authentication and integrity (but no secrecy) is provided, the user is likely to eventually type a password to another machine or service, which will then be shown in the clear.

Strong encryption seems to be the only solution to network security. Since several major governments have demonstrated growing interest in economic espionage (including, e.g., United States, Russia, France, and Japan), commercial systems are now faced with some of the most skilled and resourceful opponents in the world, and must be designed using the strongest possible methods to be of any use. Increasing economic significance will also lure interest from criminal organizations, which are certainly well enough funded to break e.g. DES-level encryption methods without much trouble [12].

2 An Overview of SSH Secure Remote Login

SSH permits secure login connections and file transfer over the Internet or other untrusted networks. Cryptographic algorithms are used to authenticate both ends of the connection, to automatically encrypt all transmitted data, and to protect the integrity of data. Values returned by services such as DNS or network protocols (e.g., TCP/IP [10]) are considered only advisory, and are validated using cryptography. SSH also automatically and securely forwards X11 connections from the remote machine, and can be configured to forward arbitrary TCP/IP ports. It can also be used for secure file transfer.

3 The SSH Protocol

SSH uses a packet-based binary protocol that works on top of any transport that will pass a stream of binary data. Normally, TCP/IP is used as the transport, but the current implementation also permits using an arbitrary proxy program to pass data to/from the server, and includes direct support for SOCKS and FWTK based firewalls.

The packet mechanism and related authentication, key exchange, encryption, and integrity mechanisms implement a transport layer security mechanism, which is then used to implement the remote login functionality. An attacker is limited to breaking the connection.

Every transmitted packet starts with random padding, followed by (optionally compressed) packet type, packet data, and integrity protection data.

The entire packet is encrypted using a suitable algorithm, such as IDEA-CFB [2, 9], 3DES-CBC [9], or an RC4[1] [9] equivalent algorithm. The packet type and data fields can be compressed with the gzip algorithm before encryption. Compression reduces the amount of transmitted data to about a third for typical interactive sessions.

Integrity protection is currently (March 1996) provided by including CRC32 [1] of the packet under encryption. However, it is being replaced by HMAC-SHA; see Section 5. If tampering is detected, the error is logged, the user is notified, and the connection is terminated.

On the transport, each encrypted packet is prefixed by the length of the packet data, excluding padding (the total length on the wire is the given length rounded up to a multiple of eight bytes in such a way that the length of padding is 1-8 bytes).

The SSH protocol works on top of the packet-level protocol, and proceeds in the following phases:

1. The client opens a connection to the server. (Note that an attacker may cause the connection to actually go to a different machine.)

2. The server sends its public RSA host key and another public RSA key ("server key") that changes every hour. The client compares the received host key against its own database of known host keys (in future, it will validate the host key using a public key infrastructure; however, at present no such infrastructure exists).

   At present, SSH is not able to validate keys for hosts that it does not already know. It will normally accept the key of an unknown host and store it in its database for future reference (this makes SSH usable in practice in most environments). However, SSH can also be configured to refuse access to any hosts whose key is not known.

3. The client generates a 256 bit random number using a cryptographically strong random number generator, and chooses an encryption algorithm from those supported by the server (normally IDEA or three-key 3DES). The client encrypts

[1] RC4 is a trademark of RSA Data Security, Inc.

the random number (session key) with RSA using both the host key and the server key, and sends the encrypted key to the server.

The purpose of the host key is to bind the connection to the desired server host (only the server can decrypt the encrypted session key). The server key is used to make decrypting recorded historic traffic impossible after the server key has been changed (usually every hour) in the event that the host key becomes compromised. The host key is normally a 1024 bit RSA key, and the server key is 768 bits. Both keys are generated using a cryptographically strong random number generator.

4. The server decrypts the RSA encryptions and recovers the session key. Both parties start using the session key (until this point, all traffic has been unencrypted on the packet level). The server sends an encrypted confirmation to the client. Receipt of the confirmation tells the client that the server was able to decrypt the key, and thus holds the proper private keys.

   At this point, the server machine has been authenticated, and transport-level encryption and integrity protection are in use.

5. The user is authenticated to the server. This can happen in a number of ways; the dialog is driven by the client which sends requests to the server. The first request always declares the user name to log in as. The server responds to each request with either "success" (no further authentication is needed) or "failure" (further authentication is required).

   Currently supported authentication methods are:

   - Traditional password authentication. The password is transmitted over the encrypted channel, and thus cannot be seen by outsiders.
   - A combination of traditional .rhosts or hosts.equiv authentication and RSA-based host authentication. Host authentication works by the server generating a 256 bit challenge, encrypting it with the client's public host key, and sending the encrypted challenge to the client. The client decrypts the challenge, and computes MD5 [7] of the challenge and other information that binds the returned value to the particular session. The client then sends this value to the server; the server makes the corresponding computations and compares the values.
   - Pure RSA authentication. The idea is that possession of a particular private RSA key serves as authentication. The server has a list of accepted public keys. The client requests authentication by a particular key, and the server responds with a challenge similar to that in RhostsRSA authentication.
   - Support is also included e.g. for Security Dynamics SecurID cards. Adding new authentication methods is easy.

6. After authentication has been successful, a preparatory phase begins. In this phase, the client sends requests that prepare for the actual session. Such requests include allocation of a pseudo-tty, X11 forwarding, TCP/IP forwarding, etc. Adding new preparatory operations is easy.

   After all other requests, the client sends a request to start the shell or to execute a given command. This message causes both sides to enter the interactive session.

7. During the interactive session, both sides are allowed to send packets asynchronously. The packets may contain data, open requests for X11 connections, forwarded TCP/IP ports, or the agent, etc. Finally at some point the client usually sends an EOF message. When the user's shell or command exits, the server sends its exit status to the client, and the client acknowledges the message and closes the connection.

More information about the protocol can be found in [13].

3.1 X11 and TCP/IP Forwarding

SSH can automatically forward the connection to the user's X server over the secure channel. Forwarding works by creating a proxy X server at the remote machine by allocating the next available TCP/IP port number above 6001 (these correspond to X display numbers so that the port corresponding to display n is $6000 + n$). The SSH server then listens for connections on this port, forwards the connection request and any data over the secure channel, and makes a connection to the real X server from the SSH client. The DISPLAY variable is automatically set to point to the proper value. Note that forwardings can be chained, permitting safe use of X applications over an arbitrary chain of SSH connections.

SSH also automatically stores Xauthority data [8] on the server. In fact, the client generates a random MIT-MAGIC-COOKIE-1 authentication cookie, and sends this cookie to the server, which stores it in `.Xauthority`. When a connection is made, the client verifies that the authority data matches the generated random data, and replaces it with the real data. The motivation for sending a fake cookie is that old cookies left at the server are useless after logout (many users keep the same terminal open for months at a time, and may briefly log into dozens of machines during that time; it is important to not leave the cookies lying around in all of these machines).

TCP/IP forwarding works similarly: the server listens for a socket on the desired port, forwards the request and data over the secure channel, and makes the connection to the specified target port from the other side. There is no authentication for forwarded TCP/IP connections.

3.2 The Authentication Agent

SSH supports using an authentication agent. The agent is a program that runs in the user's local machine (or, in future, on a smartcard connected to it). The agent holds the user's private RSA keys. It never gives out the private keys, but accepts authentication requests and gives back suitable answers.

In the Unix environment, the agent communicates with SSH using an open file handle that is inherited by all children of the agent process (the agent is started as a parent of the user's shell). Other users cannot get access to the agent, and even for root it is fairly difficult to send requests to a file descriptor held by some process. Different mechanisms are used on other operating systems.

SSH can forward the connection to the agent to another process running on the server machine (such as another SSH connection). In this way, it is possible to go through an arbitrarily long chain of machines, located anywhere around the world, without the authentication keys ever leaving the agent.

4 Cryptographic Methods Used in SSH

SSH attempts to provide strong security without making normal use any more difficult than necessary. Its security relies on cryptographic methods.

SSH uses RSA [6, 9] for host authentication and user authentication. Host keys and user authentication keys are normally 1024 bits.

The server key that changes every hour is 768 bits by default. It is used to protect intercepted historical sessions from being decrypted if the host key is later compromised. The server key is never saved on disk.

Key exchange is performed by encrypting the 256-bit session key twice using RSA. It is padded with non-zero random bytes before each encryption (according to PKCS#1 [5]). Server host authentication happens implicitly with the key exchange (the idea is that only the holder of the valid private key can decrypt the session key, and receipt of the encrypted confirmation tells the client that the session key was successfully decrypted).

Client host authentication and RSA user authentication are done using a challenge-response exchange, where the response is MD5 of the decrypted challenge plus data that binds the result to a specific session (host key and anti-spoofing cookie).

The key exchange transfers 256 bits of keying data to the server. Different encryption methods use varying amounts of the key: IDEA-CFB uses 128 bits, 3DES-CBC 168 bits, RC4-equivalent 128 bits per direction, and DES-CBC 56 bits. The reasons for using IDEA in CFB mode is mainly historical; the new protocol (Section 5) will use IDEA-CBC instead.

Transmitted data is currently protected against modification by computing a CRC32 of all packet data (including random padding) before encryption. The checksum and all packet data are encrypted. Presumably it will be difficult for an attacker to modify the plaintext data so that the checksum still matches without breaking the encryption first. (The integrity mechanism has changed in the new protocol; see Section 5.)

All random numbers used in SSH are generated with a cryptographically strong generator. SSH has a pool of 8192 bits of randomness. The first time it is started, it uses several commands to gather entropy from the system (on Unix, "ps laxww", "ps -al", "ls -alni /tmp/.", "w", "netstat -s", "netstat -an", and "netstat -in"). The entropy is mixed into the pool, stirring the pool frequently. The stirring involves encrypting the pool twice using MD5 in CBC mode so that every bit of the pool depends on every other bit. Additional noise is obtained from various system parameters (e.g., disk I/O counts, page swapping counts, interrupt counts, CPU usage) every time the pool is stirred, and if `/dev/random` is available, 128 bits of noise are taken from there every few minutes and stirred into the pool.

5 The New Protocol

The SSH protocol is currently undergoing major changes. The protocol will be split to two levels, a generic secure transport layer mechanism and a high-level SSH protocol.

5.1 The New Transport Layer Protocol

The new transport layer protocol has been designed to be flexible, allowing negotiation of all algorithms and parameters, simple, secure, easily verifiable, and fast. It performs a full algorithm negotiation, key exchange, and mutual host authentication in a total of 1.5 round-trip times typical, and 2.5 round-trip times worst case. A minimal number of round-trips will become increasingly important in future as network bandwidth increases but the speed of light remains constant. Mobile computing, in particular, will put strong demands on the number of roundtrips; over a GSM phone, for example, a round-trip is around a second.

There have been several cryptographic improvements. All data exchanged during key exchange is authenticated. HMAC-MD5 or HMAC-SHA outside encryption are used for data integrity protection. IDEA is now used in CBC mode. All data, including the packet length, is now encrypted (except the MAC). Keys are re-exchanged periodically. The protocol can also interface with a public key infrastructure.

5.2 The New SSH Protocol

The new SSH protocol runs over the transport layer protocol, which provides a secure channel. The SSH protocol performs user authentication, session management, and handshaking for multiple simultaneous connections (forwarded X11 connections, etc).

User authentication now permits the client to send authentication requests without waiting for responses from the server after each request. This reduces round-trips. Additionally, whenever an authentication request fails (or is insufficient), the server will tell the client which authentication methods can continue the dialog. This permits the server to guide the client through a multi-phase authentication according to the server's per-user policy. The server can require multiple authentications.

All authentication methods that require user input have been merged under one interactive authentication type. This handles passwords, one-time passwords, SecurID cards, and other such methods. The user basically converses with the server using a simple text-based protocol. The protocol does, however, permit dialog-based windowed implementation and local editing at the client.

The new protocol also supports proper flow control for individual channels (e.g., forwarded X11 clients). This will prevent a runaway program from jamming the entire connection. Details of the new protocol are still being specified as of this writing.

6 The Current Implementation

SSH was first published on the Internet in July 1995. Since then, it has been ported to a number of platforms and there have been several other improvements.

SSH currently runs on almost all Unix variants, including e.g. AIX, BSD, Convex, DGUX, HPUX, IRIX, Linux, Mach3, OSF/1, SysV, Solaris, SunOS, Ultrix, and Unicos. A commercial Windows version is available from Data Fellows, and a Macintosh version is due in the fall 1996. A free OS/2 version is also available.

The current Unix version supports SOCKS and FWTK based firewalls, and permits using an arbitrary proxy program to make the connection. In most environments, it can be installed simply by

```
./configure
make
make install
```

7 Performance

Performance of SSH can divided into two important parameters: startup time and transfer rate.

The startup time means the time from starting the SSH client to the moment when first data bytes are transferred. The startup time is on the order of a second on 486 or Pentium class machines connected to an ethernet, and on the order of a few seconds for long-distance connections.

Transfer rate means the number of bytes per second that can be transmitted over the secure channel. In the case of SSH, it depends on the encryption algorithm used. On 486-class machines, the rate is 1-2 megabits/second for IDEA, 3-4 megabits/second for DES, and about 5 megabits per second for RC4-equivalent. The rate is almost directly proportional to the speed of the machines; some faster machines run RC4-equivalent in software at speeds exceeding 40 megabits per second.

To summarize, the encryption speed on even slower modern machines is sufficient to fill an ethernet network. Most of the time, transfer rate is not limited

by encryption but by the transfer rate of the network. Furthermore, on long-distance connections SSH can be substantially faster than `telnet` or `rlogin`, due to compression of transferred data.

8 Conclusion

SSH solves one of the most acute security problems on the Internet: that of securely logging from one machine to another, and securely transferring files between machines. It does this in a way that is convenient and completely transparent to users. At the same time, it automates passing the X11 connection, and makes using X11 over long distance connections secure.

SSH uses strong cryptography to achieve this goal. Its fundamental principle is that the network or any of its services cannot be trusted. Usability in normal environments has been a major design concern from the beginning, and SSH attempts to make things as easy for normal users as possible while still maintaining a sufficient level of security. For the most security conscious environments, SSH can be configured to never trust the network, and fail if it cannot e.g. verify the host key of the remote host.

Experience has shown that the CPU overhead caused by strong encryption is negligible. One need not try to justify why to encrypt; doing so costs almost nothing. However, not using strong encryption in all communications can have severe consequences. Also, the strongest available encryption methods should be used, as they are no more expensive than weak methods. Weak encryption will just make transmitted data available to foreign intelligence agencies and criminal organizations.

SSH is currently (June 1996) being used at thousands or tens of thousands of sites in at least 50 countries around the world. There are about one thousand addresses on the mailing list, and many of those are redistribution aliases or newsgroup gateways. The SSH WWW pages are accessed about 1000-2000 times every day (about once every minute). During about a period of about ten days (examined in February 1996), accesses came from about 6000 hosts (many of them WWW proxy/cache servers) in 55 top-level domains. The actual number of people using SSH is not known.

SSH is freely available for non-commercial use. Its WWW home page, including pointers to ftp sites and commercial versions, is available at `http://www.cs.hut.fi/ssh`.

References

[1] J. Campbell. *C Programmer's Guide to Serial Communications*, Sams, 1993.

[2] Lai, X. *On the Design and Security of Block Ciphers.* ETH Series in Information Processing, vol. 1, Hartung-Gorre Verlag, Konstantz, 1992.

[3] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. *SOCKS Protocol Version 5*, RFC 1928, 1996.

[4] Mockapetris, P. *Domain Names - Concepts and Facilities*, RFC 1034, Internet Engineering Task Force, 1987.

[5] *Public Key Cryptography Standards, #1.* RSA Laboratories. Available for anonymous ftp at ftp.rsa.com.

[6] Rivest, R., Shamir, A., and Adleman, L. M. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, vol. 21, no. 2, 1978, pp. 120-126.

[7] Rivest, R. *The MD5 Message Digest Algorithm*, RFC 1321, Internet Engineering Task Force, 1992.

[8] Scheifler, R. *X Window System Protocol.* X Consortium Standard, Version 11, Release 6. Laboratory of Computer Science, Massachusetts Institute of Technology, 1994.

[9] Schneier, Bruce. *Applied Cryptography*, 2nd edition. John Wiley & Sons, 1996.

[10] Stevens, W. Richard. *TCP/IP Illustrated. Volume 1: The Protocols.* Addison-Wesley, 1994.

[11] TIS Firewall Toolkit, Trusted Information Systems Inc., 1993.

[12] Wiener, M. J. *Efficient DES Key Search.* Technical Report TR-244, School of Computer Science, Carleton University, 1994.

[13] Ylönen, Tatu. *The SSH (Secure Shell) Remote Login Protocol*, 1996. Available on the Internet from the SSH Home Page at http://www.cs.hut.fi/ssh. Also included in the SSH distribution.

Dual-workfactor Encrypted Key Exchange: Efficiently Preventing Password Chaining and Dictionary Attacks

Barry Jaspan*

Abstract

Password-based key-server protocols are susceptible to *password chaining* attacks, in which an enemy uses knowledge of a user's current password to learn all future passwords. As a result, the exposure of a single password effectively compromises all future communications by that user. The same protocols also tend to be vulnerable to dictionary attacks against user passwords.

Bellovin and Merrit[1] presented a hybrid of symmetric- and public-key cryptography called Encrypted Key Exchange (EKE) that cleanly solves the dictionary attack problem. This paper presents an extension of their ideas called *dual-workfactor encrypted key exchange* that preserves EKE's strength against dictionary attacks but also efficiently prevents passive password-chaining attacks.

1 Introduction

Virtually all password-based key server protocols are vulnerable to *password chaining* attacks. In these systems, a user shares only a single secret, its password, with the trusted key server. In addition to protecting the normal authentication and key-distribution messages, the current password is also used to protect the protocol exchanges used to select a new password. An enemy that already knows the user's current password can "daisy-chain" the passwords together by decrypting the password-changing session and reading out the new password.

Many password-based protocols are also vulnerable to a dictionary attack against user passwords. In the Kerberos authentication system[20], for example, the initial protocol exchange is encrypted in a key derived from the user's password. A malicious user can obtain these encrypted messages and use them to attempt to guess the user's password. Since the format of the messages is well known, a correct guess is easily verifiable. Once the guessing attack succeeds, the attacker can use the password to impersonate the user until the password is changed. In practice, password guessing attacks have a high degree of success[12, 24].

The combination of these two weaknesses is particularly worrisome. The password guessing vulnerability makes it likely that at least one of a user's passwords will eventually be disclosed; password chaining means that once one password is disclosed, all future passwords are compromised. Since it is easy for an attacker to record all of a user's traffic while simultaneously attacking old messages, both past and future communications are at risk.

Bellovin and Merrit[1] presented a fundamentally novel hybrid of symmetric- and public-key cryptography called Encrypted Key Exchange (EKE) that cleanly solves the dictionary attack problem. This paper presents an extension of their ideas called *dual-workfactor encrypted key exchange* that preserves EKE's strength against dictionary attacks but also prevents passive password-chaining attacks, without incurring unnecessarily high overhead. The basic idea is to use an extremely strong EKE exchange for password-changing messages, thus preventing chaining, while using a much weaker EKE exchange with normal authentication for efficiency. Dual-workfactor encrypted key exchange is motivated in the context of a new EKE-based pre-authentication type for Kerberos 5 called PA-ENC-DH that can be implemented without changing the current Kerberos 5 protocol or requiring additional network messages.

Section 2 discusses previous work on password chaining and dictionary attacks. Section 3 provides background on the Kerberos and Diffie-Hellman exponential key exchange protocols, and motivates dual-workfactor encrypted key exchange. Section 4 discusses implementation issues for this protocol specific to Kerberos 5, including public-key parameter selection. Table 1 summarizes the notation used throughout this paper.

*Affiliation: Independent consultant, 4 Merrill Ave, Belmont, MA 02178, *bjaspan@mit.edu*. This work was also funded in part by BBN Planet Corporation and the Internet Software Consortium.

Notation	Meaning
C	client
S	server
KDC	Key Distribution Center, a.k.a. the Kerberos server
K_x	symmetric secret key for user x
K_{xy}	symmetric session key for users x and y
$\{X\}^{K_x}$	message X encrypted in symmetric key K_x
p_x	private Diffie-Hellman key for user x
P_x	public Diffie-Hellman key for user x, $P_x = q^{p_x}$
$\{X\}^R$	message X encrypted in symmetric key R derived from Diffie-Hellman key $q^{p_c p_s}$
T_{cs}	Kerberos ticket for client c to use service s
PA $= D$	Kerberos pre-authentication with data D

Table 1: Notation used throughout this paper.

2 Previous work

There does not appear to be any previous work on strengthening password-based protocols against an attacker that already knows the password.

The issue of dictionary attacks has been widely discussed. Kerberos version 5[16] partially addresses the problem with *pre-authentication*, requiring the initial request to the KDC to prove the user's identity so an attacker cannot fraudulently request tickets for another user; see section 3.1. A protocol can be designed in which off-line guessing attacks are impractical and so every password guess must involve a central authority which can then detect the attack[18]. A password changing mechanism can be designed to enforce password quality, insisting on passwords with a minimum length, character mix, or other properties. It is also possible to design a password selection process that simplifies the selection of quality passwords that are both easy to remember and hard to guess[4].

3 Protocol description

This section provides background on the Kerberos and Diffie-Hellman protocols, explains how Encrypted Key Exchange is applied to Kerberos, and motivates dual-workfactor encrypted key exchange. The new PA-ENC-DH Kerberos pre-authentication type is presented.

3.1 Kerberos AS_REQ protocol

The most basic exchange of the Kerberos protocol is an Application Service Request, or AS_REQ. Client software uses an AS_REQ to request a Kerberos ticket for a user. The response is sent encrypted in a key derived from the user's password. Typically, an AS_REQ is used exactly once per login session to obtain a ticket-granting ticket. A simplified version is as follows:

$$\begin{aligned} C \rightarrow KDC &: C, S \\ KDC \rightarrow C &: C, T_{cs}, \{S, K_{cs}\}^{K_c} \end{aligned}$$

In the AS_REQ, the client C sends its own name and the name of a service S to the KDC. The KDC responds with an AS_REP (Application Service Reply) containing the client's name, a ticket T_{cs} for the client to use with the service, and a block of data *encrypted in the user's password-derived key* K_c containing the service name and a copy of the session key K_{cs} for the client and service to use. A dictionary attack is possible against the block encrypted in K_c in the AS_REP because the attacker can validate password guesses—if an attacker guesses K'_c, performs the decryption, and a meaningful server name S appears in the result, the guess is correct.

Kerberos 5 introduced pre-authentication into the AS_REQ exchange as a partial defense against dictionary attacks. With pre-authentication, the user's initial request is required to contain a block of data that proves the identity of the requesting user. The Kerberos server refuses to respond to the request unless the pre-authentication data is correct, thus preventing an attacker from making a fraudulent request at any convenient time. The most common pre-authentication method uses an encrypted timestamp and is called PA-ENC-TIMESTAMP:

$$\begin{aligned} C \rightarrow KDC &: C, S, \text{PA} = \{\text{time}\}^{K_c} \\ KDC \rightarrow C &: C, T_{cs}, \{S, K_{cs}\}^{K_c} \end{aligned}$$

When the KDC receives the AS_REQ, it decrypts the timestamp using the client's secret key, and only honors the request if the timestamp is within a predefined window of the KDC's current time. An attacker cannot forge the pre-authentication data without knowing K_c. However, the attacker can still read

either the request or the reply off the network and perform a dictionary attack against it. Other forms of pre-authentication exist[21] in which the request does not allow for dictionary attacks, but so far all of them leave the reply vulnerable. Thus, while existing pre-authentication schemes make a dictionary attack somewhat harder to perform, they are not a fully satisfactory solution.

3.2 Kerberos password-changing protocol

Kerberos users change their passwords by interacting with the Password Changing Service, typically implemented as part of the Kerberos administration system. The password-changing program performs an AS_REQ to obtain a ticket $T_{c,pw}$ for the administration principal S_{pw} which it then uses to perform encrypted communications with the Password Changing Server:

$$\begin{aligned} C \rightarrow KDC &: C, S_{pw} \\ KDC \rightarrow C &: C, T_{c,pw}, \{S_{pw}, K_{c,pw}\}^{K_c} \\ C \rightarrow S_{pw} &: T_{c,pw}, \{\text{new pw}\}^{K_{c,pw}} \end{aligned}$$

An attacker that already knows the user's password can decrypt $\{S_{pw}, K_{c,pw}\}^{K_c}$ to obtain $K_{c,pw}$ and then decrypt $\{\text{new pw}\}^{K_{c,pw}}$ to obtain the new password. This is called *password chaining* and it can be performed indefinitely to learn all of a user's future passwords.

3.3 Exponential key exchange

Diffie-Hellman exponential key exchange[8] is a well-known method for two parties to exchange a secret key across an open network. A good explanation appears in [23]. Briefly, the Diffie-Hellman protocol requires a prime modulus m and a primitive root of that modulus q; both values are public knowledge. Suppose that two parties C and S want to exchange a key. C picks a random number p_c, computes from it $P_c \equiv q^{p_c} \bmod m$, and sends that value to S:

$$C \rightarrow S : P_c$$

Similarly, S picks a random number p_s, computes $P_s \equiv q^{p_s} \bmod m$, and sends the value to C:

$$S \rightarrow C : P_s$$

At this point, both C and S can compute $R \equiv {P_c}^{p_s} \equiv {P_s}^{p_c} \equiv q^{p_c p_s}$. An attacker knows both P_c and P_s but cannot compute R without knowing one of p_c or p_s, neither of which is available. C and S can use R to encrypt further communications.

This protocol provides key exchange, but it does not provide authentication[9] because it is subject to a man-in-the-middle attack. An attacker can intercept all communications between C and S and substitute its own values for P_c and P_s, and neither C nor S will notice. Furthermore, the choice of m is critically important from a practical point of view[17, 22] if the protocol is to be resistant to cryptanalytic attacks.

3.4 Merging Diffie-Hellman and Kerberos

Encrypted Key Exchange (EKE) can be used to merge Diffie-Hellman exponential key exchange and Kerberos. The protocol presented here is a proposed pre-authentication type for Kerberos called PA-ENC-DH:

$$\begin{aligned} C \rightarrow KDC &: C, S, \text{PA} = (\{P_c\}^{K_c}, m) \\ KDC \rightarrow C &: C, T_{cs}, \{S, K_{cs}\}^{R}, \text{PA} = \{P_s\}^{K_c} \end{aligned}$$

The client chooses a random p_c, computes P_c, and sends P_c *encrypted with its secret key* K_c along with the modulus m as pre-authentication data. The KDC receives the request, decrypts P_c with its local copy of K_c, generates a fresh p_s and P_s of its own, and computes $R \equiv {P_c}^{p_s}$. Finally, and this is the critical part, the KDC encrypts the block containing the server name and session key in R instead of the user's key K_c. The KDC then sends the AS_REP to the client along with its own public exponential P_s encrypted in K_c. The client can decrypt $\{P_s\}^{K_c}$ to obtain P_s, use P_s and its own p_c to compute R, and use R to decrypt the KDC response and extract the session key K_{cs}.

3.5 Analysis of PA-ENC-DH

Using R to protect the AS_REP has two significant consequences. The new AS_REP is not vulnerable to a dictionary attack because R is derived entirely from random values unconnected to the user's key K_c. The new AS_REP is also not vulnerable to a passive password chaining attack, because knowledge of K_c only yields P_c and P_s which are not sufficient to compute R.

A dictionary attack against $\{P_c\}^{K_c}$ is not possible because there is no way for an attacker to verify a correct guess. P_c has no number-theoretical properties other than being less than m (this affects the choice of m; see section 4.1) and is generated from a random number p_c; thus, when an attacker guesses K'_c and gets P'_c, there is no way to tell if the guess is

right.[1] The same argument applies to P_s. Even an attacker that computes the potential P_c' for all K_c in the dictionary has still accomplished nothing because P_c alone will not decrypt the AS_REP. R cannot be computed without p_c or p_s, neither of which is available without computing the discrete log. Since both of the random exponentials are encrypted for transmission, it is extraordinarily difficult to compute the discrete log even for small values of the modulus m. If P_s were not encrypted, for example, an attacker could calculate its discrete log p_s and then validate guesses at K_c by comparing each potentially correct $P_c'^{p_s}$ with R. P_s is not available, so the enemy has to perform a dictionary attack against $\{P_s\}^{K_c}$ and then compute the discrete log of *every* result.

A man-in-the-middle attack is thwarted, as well. An attacker that intercepts $\{P_c\}^{K_c}$ cannot predict how a replacement will be decrypted by the KDC. If $\{P_c'\}^{K_{fake}}$ is inserted, the KDC will decrypt to get $P_{unknown}$ and compute $R' \equiv P_{unknown}^{p_s}$; the response will then be encrypted in R' but will include the legitimate P_s. Since the attacker cannot replace P_s without knowing K_c, the client will compute $R \equiv P_s^{p_c} \not\equiv R'$, and will notice the attack when it is unable to decrypt the response.

An attacker cannot successfully forge a response from the KDC because he cannot generate a correct pair $(\{P_s\}^{K_c}, R)$ matching the client's p_c. Furthermore, since the exchanged key R in the reply is derived from the value P_c just provided to the KDC, the client is assured that the KDC's response is fresh and not replayed. This guaranteed freshness has the additional benefit that the client can set its clock by the ticket's timestamp, using the KDC itself as a secure time service and eliminating Kerberos' dependence on time synchronization[6]. This is a simple variation on the basic idea presented in [7].

Finally, even if the attacker already knows K_c, he still cannot determine R without performing a discrete logarithm computation. Without R, the attacker cannot obtain the session key K_{cs} so as to eavesdrop on the conversation. It is this property that prevents passive password chaining attacks—if the modulus m used to compute R is long enough, knowledge of the current password cannot be used to discover future passwords.

3.6 Dual-workfactor encrypted key exchange

The previous analysis of PA-ENC-DH presents a conflict in choosing the modulus length. For efficiency, the modulus should be as short as possible because the KDC's throughput will generally be limited by the time required to perform exponentiations modulo m. The nature of PA-ENC-DH allows the modulus to be considerably shorter than that which would normally be required for long-term Diffie-Hellman security. However, a short modulus limits the effectiveness of one of PA-ENC-DH's primary benefits, that knowledge of the password alone cannot be used to learn the session key K_{cs} without also computing a discrete logarithm modulo m. If a short modulus is used and the user's password is ever compromised, any individual recorded session protected by that password can be disclosed merely by computing a single discrete logarithm under the small m. This is a substantial improvement over the current Kerberos protocol, but does not fully take advantage of PA-ENC-DH's potential.

The solution is to enhance EKE by using *dual-workfactor* encrypted key exchange, employing a harder public-key problem for the more critical password-changing messages than for the day-to-day authentication messages. In particular, the password-changing Kerberos exchange should be protected with a modulus that provides sufficiently long-term protection for the password-changing message all by itself, perhaps a value 1,024 or 2,048 bits in length. An attacker with the user's password will be able to read previously recorded ordinary sessions protected with the shorter modulus of perhaps a few hundred bits with only moderate difficulty, but when the attacker comes to the password-changing session the discrete logarithm will not be computationally feasible. The exposure of a disclosed password will thus be limited to the sessions actually protected by that password. Meanwhile, the majority of AS_REQs used to acquire normal ticket-granting tickets can still use a smaller modulus for efficiency.

4 Implementation issues

This section discusses some implementation requirements for the PA-ENC-DH extension to the Kerberos 5 protocol, including a discussion of Diffie-Hellman parameter selection. Familiarity with the discrete logarithm problem and Kerberos is assumed.

4.1 Properties of m and q

Any modulus selected must conform to all the normal requirements for a good prime for the discrete logarithm problem. The current literature[15, 26] recommends using *safe primes* for which both m and $(m-1)/2$ are prime. These primes are time-

[1] Note that this is only true if P_c is not encoded with redundant information (e.g. with ASN.1[5]) before encryption.

consuming to find but, as described below, can be computed off-line.

PA-ENC-DH imposes the additional requirement on m that it not provide an attacker any information that can be used to validate a guess at a user's password; this is not normally a consideration with Diffie-Hellman. Such information can be leaked because, with an N-bit m, the random exponentials P_c and P_s by definition satisfy

$$P_x < m < 2^N$$

but the decryption of $\{P_c\}^{K_c}$ with an incorrect guess of K_c can have any value less than 2^N with equal probability. A guess K_c' that yields a value

$$m \leq P_c' < 2^N$$

can be immediately discarded as incorrect, reducing the number of entries in the dictionary for which an attacker must compute a separate discrete logarithm. Viewed another way, a guess of K_c' that yields a valid P_c' for a large number of messages has an improved probability of being correct; in effect, this is a simple dictionary attack applied over multiple messages instead of a single message.

This probabilistic attack can be prevented by choosing m to be close enough to its maximum value 2^N. [2] How close is enough? The probability that a guess K_c' will yield a P_c' smaller than m, and thus be possibly correct, is $\frac{m}{2^N}$. Thus each message will allow an attacker to reduce the size of the working dictionary for that user by a factor of $\frac{m}{2^N}$; the fraction $1 - \frac{m}{2^N}$ of passwords in the dictionary will be disqualified. The effect is cumulative, so if an attacker steals M messages, the working dictionary will be reduced by a factor of $(\frac{m}{2^N})^M$[6].

If the attacker's dictionary has D entries and it takes T time to compute a single discrete logarithm in the field, m must be chosen so that $D * T * \frac{m}{2^N}^M$ is still large enough to make the attack impractical. It is reasonable to assume that an attacker could collect on the order of 1,000 messages encrypted in K_c—two per day during login, assuming the password is changed only annually. If m differs from 2^N by no more than one part in 10,000, the attacker's work load will be reduced by a factor of $(0.9999)^{1000} \approx 0.90$, or by about 10%, which seems reasonable. Consequently, m should be chosen so that its first 14 bits are all 1.[3]

If the modulus m is a safe prime, the generator q can be 2; the provides an additional speed advantage[26].

4.2 Generation of exponents

The most recent work on how long Diffie-Hellman exponents need to be in order to provide sufficient security[26] confirms the widely held opinion that the exponents should be twice as long as the symmetric key that will be derived from the exponentially exchanged key. For Kerberos, the longest keys currently in use are 168 bits, for Triple-DES. Thus, the exponents need be at least 336 bits long. Note that if the exponentials are not a multiple of the symmetric cryptosystem's block size in length, the extra high-order bits of the last encrypted block of $\{P_x\}^{K_c}$ must be filled with random data[1]; alternatively, the exponentials' length can simply be rounded up.

The exponents p_c and p_s selected by the client and KDC must be "random" in order for PA-ENC-DH to be secure. The question of how much randomness is needed and how it can be generated on a computer without a hardware random number source is still unresolved. [11] and [14] address this issue and provide various implementation suggestions. The implementation must be careful not to use the client's password K_c to generate random exponents in a way that will expose it to an indirect dictionary attack. For example, a guessable seed for a PRNG whose output is encrypted with K_c will provide at least some verifiable bits of plaintext to an attacker.

4.3 Modulus length recommendations

To perform a dictionary attack against PA-ENC-DH, an enemy must compute a discrete logarithm for every password guess. The cost of a dictionary attack is therefore the product of the time to compute the logarithm and the size of the attacker's dictionary. The best algorithms for computing discrete logarithms are executed in two phases, a pre-computation phase and a per-exponent phase. The pre-computation phase involves building a large, modulus-specific database and is the time consuming portion of the attack. The per-exponent phase is relatively fast. In fact, [17] estimates that discrete logarithms for a 192 bit modulus can be computed in only several minutes after pre-computation is complete. If "several minutes" for a

[2] Bellovin and Merritt suggest an alternative, and perhaps simpler, solution. Instead of transmitting $\{P_c\}^{K_c}$, transmit $\{P_c + rm\}^{K_c}$ where rm is added to P_c using non-modular addition such that the value encrypted can have any value $0 < P_c + rm < 2^N$ with equal probability; the legal values of r are discussed in [1]. The legitimate user can be sure the decrypted value is right and thus can factor out rm after decryption; the attacker doesn't know whether the decrypted value is greater than m because of the addition or because the guessed password is wrong and so gains no information.

[3] The number of most-significant bits of $2^N - 1$ and $p(2^N - 1)$ that are equal is $\lceil N - 1 - \log_2(2^N - 1 - p(2^N - 1)) \rceil \approx \lceil N - 1 - \log_2((2^N - 1)(1 - p)) \rceil \approx -\lceil \log_2(1 - p) \rceil$. For $p = 0.9999$, $-\lceil \log_2 0.0001 \rceil = 14$ bits.

192 bit modulus actually means five minutes, it would take just less than a year to try 100,000 passwords, after pre-computation. Depending on site password policies and the importance of past sessions remaining secret, that may not be long enough. As described in section 4.2, the modulus must be at least 336 bits anyway. This might be sufficient, but it seems prudent to use a 512 bit modulus for now. [15] states that pre-computation on a 512 bit modulus currently takes about a year. Assuming per-exponent computation time grows at least linearly with length, an attacker would then have to spend at least several more years guessing passwords. This is probably strong enough protection for a user that might choose a guessable password.

The situation is more clear for the password-changing modulus. The modulus must be chosen with the assumption that an attacker already knows the user's password and, therefore, the public exponentials. Only one discrete logarithm computation will be required. Therefore, the modulus must be long enough to prevent the pre-computation phase entirely. [15] suggests that 1,024 bits is enough to protect a single modulus for a decade, and a 2,048 bit modulus is currently believed to prevent discrete logarithm computation forever. Since passwords are typically changed at most monthly and often much less frequently than that, a reasonably high delay can be tolerated, so there is no good reason not to use 2,048 values.

4.4 Configuration of moduli

The specific modulus to use for each transaction is not a fixed constant for the protocol but rather is specified by the client as part of the protocol message itself. This is necessary to allow dual-workfactor encrypted key exchange. It also frees the protocol from being tied to a specific modulus which may be found to be too short or too weak for other reasons.

Moduli can be site-specific and changed as often as necessary to keep up with computational speed increases and theoretic developments. The various moduli a client is allowed to use should be specified in a configuration file (e.g. `/etc/krb5.conf`) that is available on all client hosts.[4] Two classes of moduli must be specified, one for normal authentication and one for use with the password-changing protocol. The client can simply choose one of the listed values, and then include it in the PA-ENC-DH message; alternatively, the protocol could be designed to specify an index into a list of moduli for transmission efficiency. The KDC should have a copy of the same list of moduli and accept a message based on any of them, but refuse a message based on any other value on the grounds that it may not be secure.

[4] Clients must verify that the moduli meet the requirements of section 4.1 to prevent trojan horse values.

5 Consequences and limitations of PA-ENC-DH

PA-ENC-DH only prevents *passive* password chaining attacks against recorded sessions. An attacker that knows the user's password and fully controls the network (rather than merely being able to listen on the network) can perform an *active* man-in-the-middle attack by replacing the encrypted exponentials in the client's and server's message with its own value. The attacker can then allow the client and server to complete the password change without interruption while at the same time learning the user's new password. Interestingly, it seems *almost* possible for a client to detect a man-in-the-middle attack by timing the password-changing protocol, since an attacker is forced to compute several exponentiations himself; however, a clever attacker can avoid such detection[6].

PA-ENC-DH does not actually provide pre-authentication of the client's request to the Kerberos server. Furthermore, it makes using other forms of pre-authentication questionable since some of them (e.g. PA-ENC-TIMESTAMP) will re-expose the client's password to a dictionary attack. This implies that it is once again not possible to have any control whatsoever on who requests tickets for a given user. As PA-ENC-DH solves the problem that pre-authentication was designed to solve better than pre-authentication does, this is not really a drawback.

The lack of pre-authentication does expose PA-ENC-DH to an undetectable on-line password guessing attack[10]. Since the KDC cannot authenticate the entity performing an AS_REQ or detect the freshness of a request, an attacker is free to make password guesses by performing the AS_REQ protocol with PA-ENC-DH pre-authentication at any time. This is not a serious threat, however. The KDC can be modified to allow only a certain number of requests per day and refuse (or simply log to a high-priority channel) any requests over the limit. An upper limit of perhaps 20 requests per day will be more than enough for users but still low enough to prevent such on-line guessing attacks.

The KDC is sometimes used to maintain state about user logins (e.g. invalid login attempts, last login time) that provide useful administrative information. Pre-authentication allowed this information to

be considered "trustworthy" because a request with bad pre-authentication data would be rejected without updating user state. Without pre-authentication, the KDC cannot tell a valid request from an invalid one. However, most of this information can be tracked just as usefully without pre-authentication—there is no loss in security, for example, of considering the last login time of a user to be the time of the last AS_REQ, authenticated or not.

6 Conclusion

With only a single modulus, EKE can either prevent password chaining but make the protocol slow, or prevent dictionary attacks without protecting against password chaining. Dual-workfactor encrypted key exchange allows EKE to be used efficiently to prevent both of these attacks.

PA-ENC-DH simultaneously solves three long-standing limitations of Kerberos: vulnerability to dictionary attacks, dependence on insecure time synchronization, and vulnerability to password chaining. There are other solutions for at least the first two of these problems, notably password quality and secure time protocols, but none of them solve all three problems simultaneously and with such simplicity. Although other public-key modifications of the Kerberos protocol have been proposed that accomplish some or all of these goals, only PA-ENC-DH does so without introducing extra protocol exchanges or requiring substantial protocol modification. PA-ENC-DH also preserves the primarily symmetric-key nature of Kerberos, giving it both higher performance and better long-term security prospects than purely public-key based systems.

The PA-ENC-DH protocol can be varied and generalized in many ways. It applies to essentially any password-based key server protocol, not just Kerberos. Any number of different workfactors can be employed to provide an appropriate level protection for multiple categories of transactions. Diffie-Hellman exponential key exchange can be replaced with many other public-key cryptographic algorithms, notably including elliptic curve cryptography, which may provide substantial practical benefits such as greater resistance to cryptanalysis.

Naturally, PA-ENC-DH should not be used as an excuse to lessen the emphasis on choosing quality passwords nor on protecting those passwords carefully. A password-based protocol will still be more secure with PA-ENC-DH and a good password than with PA-ENC-DH and a bad password.

7 Acknowledgements, patents

Don Davis and Brian LaMacchia reviewed early drafts of this paper and provided substantial helpful discussion and analysis of PA-ENC-DH. Don also introduced me to Bellovin and Merrit's previous work in this area after reviewing an initial sketch of this paper.

Diffie-Hellman exponential key exchange is covered by a U.S. patent[13] which expires in March, 1997. Bellovin and Merrit's Encrypted Key Exchange is covered by a U.S. patent[3] which expires in August, 2010.

References

[1] Steven M. Bellovin and Michael Merrit. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, May 1992.

[2] Steven M. Bellovin and Michael Merritt. Limitations of the Kerberos authentication system. In *USENIX Conference Proceedings*, pages 253–267, Dallas, TX, Winter 1991. USENIX. A version of this paper was published in the October, 1990 issue of Computer Communications Review.

[3] Steven M. Bellovin and Michael Merritt. Cryptographic protocol for secure communications. U.S. Patent 5,241,599, August 1993.

[4] Matt Bishop. Anatomy of a proactive password checker. In *Proceedings of the 3rd USENIX UNIX Security Symposium*, Baltimore, MD, September 1992.

[5] CCITT. Recommendation X.509: The Directory Authentication Framkework, December 1988.

[6] Don Davis. Personal communication.

[7] Don Davis, Daniel Geer, and Theodore Ts'o. Kerberos with clocks adrift: History, protocols, and implementation. In *Proceedings of the 5th USENIX UNIX Security Symposium*, Salt Lake City, June 1995.

[8] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Trans. Inform. Theory*, IT-22:644–654, November 1976.

[9] Whitfield Diffie, Paul C. Van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes, and Cryptography*, 2(2):107–125, June 1992.

[10] Yun Ding and Patrick Horster. Undetectable online password guessing attacks. *Operating Systems Review*, 29(4):77–86, October 1995.

[11] D. Eastlake, S. Crocker, and J. Schiller. RFC 1750: Randomness Recommendations for Security, December 1994.

[12] David C. Feldmeirer and Philip R. Karn. UNIX password security - ten years later. In G. Brassard, editor, *Proc. CRYPTO 89*, pages 44–63. Springer-Verlag, 1990. Lecture Notes in Computer Science No. 435.

[13] M. Hellman and R. C. Merkle. Public key cryptographic apparatus and method. U.S. Patent 4,218,582, August 1980.

[14] IEEE P1363 working group. Standard for RSA, Diffie-Hellman, and Related Public Key Cryptography: Appendix E, November 1995. This is a draft document currently available at http://stdsbbs.ieee.org/groups/1363.

[15] P. Karn and W. A. Simpson. The Photuris session key management protocol, November 1995. This is a working draft document of the IETF.

[16] John Kohl and Clifford Neuman. RFC 1510: The Kerberos Network Authentication Service (V5), September 1993.

[17] B.A. LaMacchia and A.M. Odlyzko. Computation of discrete logarithms in prime fields. In A.J. Menezes and S. A. Vanstone, editors, *Proc. CRYPTO 90*, pages 616–618. Springer-Verlag, 1991. Lecture Notes in Computer Science No. 537.

[18] T. Mark A. Lomas, Li Gong, Jerome H. Saltzer, and Roger M. Needham. Reducing risks from poorly chosen keys. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 14–18, December 1989.

[19] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[20] B. Clifford Neuman and Theodore Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, September 1994.

[21] Clifford Neuman and Glen Zorn. Integrating single-use authentication mechanisms with Kerberos, November 1995. This is a working draft document of the IETF.

[22] S. C. Pohlig and M. E. Hellman. An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance. *IEEE Trans. Inform. Theory*, IT-24:106–110, January 1978.

[23] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* John Wiley & Sons, New York, 1993.

[24] Eugene Spafford. Observing reusable password choices. In *Proceedings of the 3rd USENIX UNIX Security Symposium*, Baltimore, MD, September 1992.

[25] J.G. Steiner, B.C. Neuman, and J.I. Schiller. Kerberos: an authentication service for open network systems. In *Usenix Conference Proceedings*, pages 191–202, Dallas, Texas, February 1988.

[26] P. C. van Oorschot and M. J. Wiener. On Diffie-Hellman key agreement with short exponents. In *Proceedings of Eurocrypt '96* (to appear). Springer-Verlag, 1996.

Security Mechanism Independence in ONC RPC

Mike Eisler, SunSoft, Inc., mre@eng.sun.com
Roland J. Schemers III, Stanford University, schemers@leland.stanford.edu
Raj Srinivasan, ControlNet, Inc., rajsrin@aol.com

(Note: Roland Schemers and Raj Srinivasan contributed to this work while employed at SunSoft, Inc.)

ABSTRACT

Generic Security Services API (GSS-API) [4] provides a framework for security services. It allows source level portability. It allows applications to run independently of the underlying security mechanisms and technologies.

To provide security mechanism independence in ONC* RPC [1, 2, 3], this paper proposes a new security flavor, RPCSEC_GSS. RPCSEC_GSS incorporates services offered by the GSS-API into ONC RPC. Using the programming interface for the RPCSEC_GSS flavor, ONC RPC applications can specify a GSS-API security mechanism to be used with an RPC session, and also request security services, such as integrity and privacy.

INTRODUCTION

This document describes the proposed security architecture for ONC RPC. The reader of this document should understand the GSS-API and be familiar with ONC RPC.

REQUIREMENTS

The security framework proposed in this document is driven by the following requirements:

- The framework must support multiple underlying security mechanisms, and provide access to their services through a common API. Applications using this API should not be required to be aware of the underlying security mechanism.
- The API should be flexible enough to enable different applications to meet their security requirements, while still being easy to use.
- The framework should enable ISVs to add new security mechanisms in a well-defined way. (Note that the actual implementation may not allow for this, to comply with government export control restrictions.)
- The framework should accommodate existing applications by providing for binary and source compatibility. That is, existing binaries and source code should continue to work as before.
- Components used to implement this framework should not be specific to ONC RPC. As far as possible, components, such as the GSS-API, should be available for use by applications that don't require ONC RPC services.

ONC RPC FLAVORS

The Solaris 2.x *Network Interfaces Programming Guide* describes how security flavors work in ONC RPC.

To date, security flavors have provided only authentication services of varying strengths. The AUTH_DES and AUTH_KERB (version 4) security flavors use cryptography to provide stronger authentication than the AUTH_SYS security flavor, which does not use cryptography and therefore can be more easily spoofed.

Historically, security flavors have been called *authentication flavors*. This is probably because integrity and privacy were not considered requirements. Also, the computational overhead of integrity and privacy was considered prohibitive for an application like the NFS† environment (which drove RPC architecture for the main part). Today, no fundamental reason exists for restricting new security flavors in the services they perform.

Security Mechanisms vs. Security Flavors

Examples of security mechanisms are Kerberos V5, RSA public key, Diffie-Hellman public key, PGP, etc. A security mechanism specifies cryptographic techniques to achieve data authentication or confidentiality. Usually, a security mechanism does not specify transport protocols or conventions; security mechanisms can be used in conjunction with different data transport protocols. For example, the Kerberos V5 [6, 9] mechanism specifies certain conventions and formats that allow it to be used with ONC RPC, FTP, TELNET, or DCE RPC.

Security flavors are specific to ONC RPC and are administered by `iana@isi.edu`. Each security flavor is represented by a unique integer and is carried in the RPC protocol header. A security flavor indicates how to interpret RPC security headers (the credential and verifier fields) and client data. Until now, since integrity and privacy have not been used, client data formats have been the same for all security flavors. Also, existing security flavors embody different security mechanisms, and each security flavor has its own API. Currently, ONC RPC applications that require security services use security flavors that embody specific security mechanisms. These applications have to be modified to make use of other security mechanisms.

* ONC is a trademark of Sun Microsystems, Inc.

† NFS is a trademark of Sun Microsystems, Inc.

THE GSS-API FRAMEWORK

The GSS-API framework defines a generic security service standard and an API that offers the same security services to applications, while using multiple underlying security mechanisms. The GSS-API framework is described in [4] and its C language bindings are described in [5]. This API framework has some similarities to TLI. It "normalizes" accesses to different security mechanisms, so that applications using the GSS-API need not be aware of underlying security mechanisms.

Figure 1 illustrates this concept:

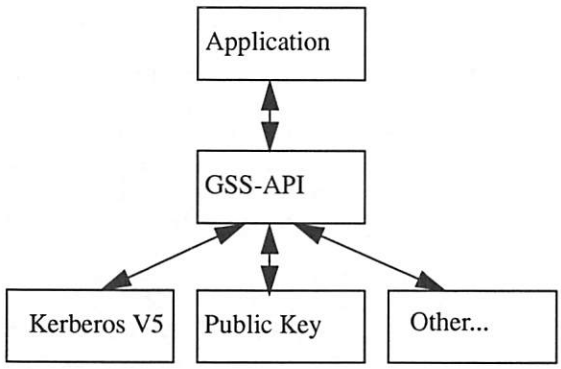


Figure 1 GSS-API Framework

The basic services offered by the GSS-API are integrity and privacy. In the integrity service, the GSS-API uses the underlying mechanism to authenticate messages exchanged between applications. Cryptographic checksums establish the identity of the data originator to the receiver, the identity of the receiver to the originator (if mutual authentication is requested), and the authenticity of the transmitted data. The privacy service includes the integrity service. In addition, the transmitted data is encrypted to protect it from any eavesdroppers.

The GSS-API does not specify any transport mechanisms. Calls to the GSS-API primitives return tokens, which the application must convey to its peer. For example, if the application is ONC RPC, clients will convey the tokens created by the GSS-API to servers and vice-versa, in RPC messages. These tokens may represent entities such as cryptographic checksums or encrypted data.

Before an application begins to exchange data with a peer, it must first establish a context for the exchange. The GSS-API primitive that creates the context requires a parameter that specifies the security mechanism to be used for that context. Applications that wish to be independent of the security mechanism must provide this parameter transparently, by obtaining it either from a configuration file, or the name service.

Once a context is established, applications can begin to exchange data units. For each data unit, the application can specify whether the integrity service or the privacy service or neither applies. The GSS-API defines these services. They are independent of the security mechanism. The services may be determined dynamically or hard-coded by the application without compromising security mechanism independence.

For each data unit, the application can also specify the Quality of Protection (QOP) to be used. The QOP parameter determines the cryptographic algorithms to be used in conjunction with the integrity or privacy service. This parameter is security mechanism dependent. Therefore, to be independent of the underlying security mechanism, an application has to either obtain the QOP parameter from a configuration file or name service, or use the default QOP for the security mechanism.

RPCSEC_GSS

This paper proposes a new security flavor, RPCSEC_GSS, to meet the requirements stated earlier. The RPCSEC_GSS security flavor incorporates services offered by the GSS-API into ONC RPC. Figure 2 illustrates the layering of the RPCSEC_GSS security flavor interfaces above the GSS-API:

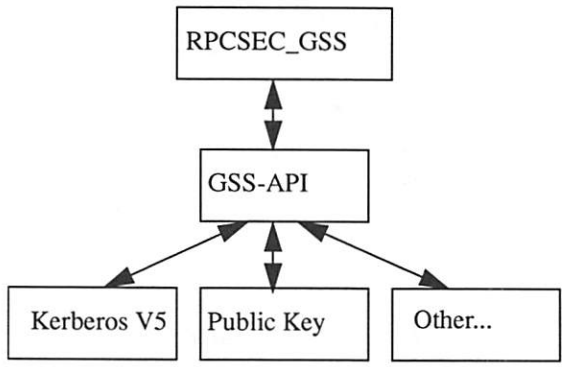


Figure 2 RPCSEC_GSS security flavor

Using the programming interface for the RPCSEC_GSS security flavor, ONC RPC applications can specify a security mechanism to be used on an RPC session, and request desired security services (integrity, privacy, or none) for each RPC exchange. Applications can also specify the desired QOP for each RPC exchange. As noted before, if the application needs to be independent of the security mechanism, it should not hard code security mechanism names or QOP values. These parameters will be passed through the RPCSEC_GSS layer to the GSS-API layer.

Since the RPCSEC_GSS security flavor uses the GSS-API interface, any security mechanism supported by the GSS-API will be available to ONC RPC applications via the RPCSEC_GSS security flavor interface. Thus, in some sense, this will be the "last security flavor" that ONC RPC needs to support to use multiple underlying security mechanisms. Moreover, adding a new security mechanism to the existing set supported by the GSS-API will not require an ONC RPC application to be modified for use with the new security mechanism. Of course, if a security

mechanism is created that offers different services than those provided by the GSS-API, and that cannot be supported under the GSS-API, it also will not be available with the RPCSEC_GSS flavor. The new security mechanism will require a different API to capture the sense of the different services it offers.

The programming paradigm for the RPCSEC_GSS security flavor is similar to that for the existing security flavors. The additional complexity is due to the extra services and options available.

THE RPCSEC_GSS PROGRAMMING INTERFACE

The following section describes the programming interface for the RPCSEC_GSS security flavor, followed by some usage examples.

This programming interface is the same for kernel and user-level RPC implementations, unless indicated otherwise.

The following definitions are used:

```
#define MAX_GSS_MECH 64
#define MAX_GSS_NAME 128

/* security service types */
typedef enum {
    rpc_gss_svc_default = 0,
    rpc_gss_svc_none = 1,
    rpc_gss_svc_integrity = 2,
    rpc_gss_svc_privacy = 3
} rpc_gss_service_t;

/* GSS input options */
typedef struct {
    int req_flags;
    int time_req;
#ifdef _KERNEL
    cred_t *my_cred;
#else
    gss_cred_id_t my_cred;
#endif
    gss_channel_bindings_t
        input_channel_bindings;
} rpc_gss_options_req_t;

/* GSS output options */
typedef struct {
    int major_status;
    int minor_status;
    int rpcsec_version;
    int ret_flags;
    int time_ret;
    gss_ctx_id_t gss_context;
    char actual_mechanism[MAX_GSS_MECH];
} rpc_gss_options_ret_t;
```

Creating a Security Context

The `rpc_gss_seccreate` function shown below is used to create a security context.

```
AUTH *rpc_gss_seccreate(
    CLIENT *clnt,          /* in */
    char *principal,       /* in */
    char *mechanism,       /* in */
    rpc_gss_service_t service_type, /* in */
    char  *qop,            /* in */
    rpc_gss_options_req_t
        *options_req,  /*in */
    rpc_gss_options_ret_t
        *options_ret);/* out */
```

`rpc_gss_seccreate()` returns a security context handle (the authentication handle). The parameters of this function are:

- `clnt` – Client handle. Context creation may involve an exchange with the server. This is why a client handle incorporating a transport connection to the server is passed as a parameter.
- `principal` – Identity of the server principal. The name has the form *<service>@<host>*, where *<service>* is the name of the service that the client wishes to access; *<host>* is the fully qualified name of the host where the service resides, for example, nfs@jurassic.eng.sun.com.
- `mechanism` – An ASCII string, for example, "kerberos_v5," that results in the use of the appropriate GSS-API backend. This mapping is done via a configuration file. The mechanism may be obtained either from the client program's configuration, or as a result of a name service lookup. This ASCII mechanism name space will be managed in ONC RPC. Note that due to export control restrictions, available mechanisms may be limited to those statically linked in with the GSS-API library.
- `service_type` – Specifies the type of service requested on the session. The `service_type` sets the initial service option. This can be changed in the course of a session. For example, when NIS+ replicates, most of the information needs to be only integrity protected, but some tables such as the password table, or other tables which are not world readable, may be privacy protected; to do so, the initiator may change the service for the session.
- `qop`: Determines the initial quality of protection for messages exchanged in this session. `qop` is specified as an ASCII string. If no QOP is specified, the default QOP for the mechanism will be used. An example of a QOP string is "GSS_KRB5_CONF_C_QOP_DES" [9].
- `options_req` – Options passed through directly to the GSS-API. These are described in the GSS-API specification. Note that the kernel version of these options uses `cred_t` (UNIX credentials) rather than `gss_cred_id_t` for delegated credentials. This is because `cred_t` is a more natural formulation for the kernel. For example, NFS will act on behalf of users whose UNIX credentials it already has. If `options_req` is NULL, defaults will be used.
- `options_ret` – Options returned to the caller. These are described in the GSS-API specification. The caller may specify NULL if it is not required to see these options.

Destroying a Security Context

The usual call:

```
void auth_destroy(AUTH *auth);
```

is used to destroy a context.

Setting the Server Principal Names

The server needs to be told the names of the principals it will be representing when it starts up. A server may act as more than one principal. The following call sets the server's principal name:

```
bool_t
rpc_gss_set_svc_name(
    char    *principal;
    char    *mechanism;
    u_int   req_time;
    u_int   program;
    u_int   version);
```

The parameters are as follows:

- `principal` –The server's principal name, specified in the *<service>@<host>* form.
- `mechanism` – Mechanism for which the name is registered.
- `req_time` – Time during which the credential should be valid.
- `program` – RPC program number for the service.
- `version` – Version number for the service.

If a server wishes to act as multiple principals, it may invoke `rpc_gss_set_svc_name` as many times as necessary.

Receiving Credentials at the Server

The `rpc_gss_getcred` function is used to obtain the client's credentials.

```
/* pointer to (opaque) principal type */
typedef struct {
    int     len;
    char    name[1];
} *rpc_gss_principal_t;

/* raw credentials */
typedef struct {
    int     version;
    char    *mechanism;
    char    *qop;
    rpc_gss_principal_t client_principal;
    char    *svc_principal;
    rpc_gss_service_t service;
} rpc_gss_rawcred_t;

/* unix credentials */
typedef struct {
    uid_t   uid;
    gid_t   gid;
    short   gidlen;
    gid_t   *gidlist;
} rpc_gss_ucred_t;

int rpc_gss_getcred(
    struct svc_req *req, /* in */
    struct rpc_gss_rawcred_t **rcred, /*out*/
    struct rpc_gss_ucred_t **ucred, /* out */
    void **cookie); /* out */
```

`rpc_gss_getcred()` allows the server to fetch the credentials of the client. Its parameters are:

- `req` – Received request, input to `rpc_gss_getcred()`.
- `rcred` – Pointer to an `rpc_gss_rawcred_t` structure pointer. The calling and called principal names are returned. The GSS mechanism is also returned, because an application may wish to determine how "strong" the security mechanism over which the request was received is, and refuse service if the mechanism is not strong enough. The QOP used for the message is returned for the same reason. The service parameter indicates whether the request was integrity protected or privacy protected. The server may choose to refuse service for a request, for example, if the message does not have privacy protection. If an application is not interested in these details, it may pass NULL for the `rcred` parameter.
- `ucred` – Caller's UNIX credentials. The application may pass NULL if it does not require these. If a client does not have UNIX credentials, a NULL pointer will be returned in this field.
- `cookie` – Pointer to a `(void *)`. If not NULL, the value returned will be that specified by the server when the context was created. Cookies are specified through the server callback mechanism described below.

Note that the credentials returned need not be destroyed or deallocated because they are pointers to entities within the security context. The application should not modify them.

Server Callbacks

A server may need to specify a callback so that it knows when a context gets established. This callback may be specified through the `rpc_gss_set_callback()` call:

```
typedef struct {
    u_int  program;
    u_int  version;
    bool_t (*callback)();
} rpc_gss_callback_t;

bool_t
rpc_gss_set_callback(rpc_gss_callback_t *cb);
```

The callback routine will be invoked any time a context is established for the specified program and version. This routine must be MT safe, since contexts can be established concurrently. This routine should return TRUE if the context is to be accepted, and FALSE otherwise:

```
bool_t callback(
    struct svc_req *req,
    gss_cred_id_t  deleg,
    gss_ctx_id_t   gss_context;
    rpc_gss_lock_t *lock,
    void           **cookie);
```

The `rpc_gss_lock_t` structure is defined as follows:

```
typedef struct {
    bool_t             locked;
    rpc_gss_rawcred_t *raw_cred;
} rpc_gss_lock_t;
```

The server can receive any delegated credentials at this point. The server also gets a pointer to the GSS context in case it wants to do GSS operations on the context to test for acceptance criteria. The per-message information in the `raw_cred` structure (qop, service) will be set to the values specified by the initiator for these parameters in the `rpc_gss_seccreate()` call.

The server also gets a pointer to the RPC request associated with the context initiation request. This will enable the server to determine the network address of the initiator, for example, by using existing calls.

The `lock` parameter enables the server to enforce a particular QOP and service for the session. When the callback is invoked, the lock parameter will point to a structure whose type is `rpc_gss_lock_t`. The `locked` field in this structure will be set to FALSE. The server should set `locked` to TRUE, to lock the context. A context that is "locked" in this way will reject all requests that have different values for service or QOP than those specified in the `raw_cred` structure. (These should not be changed by the server.)

The server can also specify a cookie that will be returned along with the caller's credentials for each request (in the `rpc_gss_getcred()` call). This cookie can be used in anyway appropriate for the application - RPC will not interpret this quantity. For example, the cookie may be a pointer or index to a structure that represents the context initiator. Instead of computing this for every request, the server does this at context creation time, and saves on request processing time.

If the server does not specify a callback, all incoming contexts will be accepted.

Changing Default Parameters for a Session

The `rpc_gss_set_defaults` function changes certain default parameters for a session.

```
int rpc_gss_set_defaults(
    AUTH *auth,
    rpc_gss_service_t service,
    char *qop);
```

This call applies to the client side only. The application may choose to switch from integrity protection to privacy protection. The application may also wish to change the QOP. The new values set by the application will apply to the rest of the session (unless changed again).

Principal names

Principal names are required in two different circumstances:

1. When clients need to specify the server's principal name, and,

2. When a server needs to operate on a client's principal name (for example, to check access control lists).

A server's principal name is always specified as a NULL-terminated ASCII string in the form *<service>@<host>*. Here, *<service>* is the name of a service, for example, `nfs`, and *<host>* is the fully qualified DNS name of a host, for example, `jurassic.eng.sun.com`. The GSS-API Version 2 specification allows such names to be used with any underlying security mechanism.

The principal name of a client as received by a server has the structure corresponding to the `rpc_gss_principal_t` defined in Receiving Credentials at the Server on page 4. This is a counted, opaque byte string. In general, the format of this string will be mechanism specific. These strings can be used either as database indices for looking up a UNIX credential (if one exists), or directly in access control lists. Now a server may wish to compare a principal name it has received with the principal name of a known entity. To do so, the server needs to generate these principal names for known entities.

The `rpc_gss_get_principal_name()` call takes as input a mechanism type, and parameters that uniquely identify an individual in a network. It returns a principal name in mechanism-specific format, which may be byte-compared with received principal names (for example, as part of a client's network credentials).

It is hard to pin down the criteria for all security mechanisms that will uniquely identify an individual in a network. For now, the following parameters seem to be sufficient: an individual or service name, a node name, and a security domain. An individual name could be a UNIX login identifier. An example of a service name is "nfs." An example node name is a UNIX machine name, such as jurassic. A security domain could be a DNS, NIS, or NIS+ domain name.

The principal name constructor will accept NULLs for one or more of the three parameters (name, node, security domain) that identify an individual in a network. For each security mechanism, the parameters necessary to construct principal names will be specified. For example, Kerberos V5 will require: a user name; optionally, a fully qualified host name; and optionally, a realm name. The construction of the principal name is mechanism dependent. Some mechanisms may do this algorithmically, and some may look up the principal name in a name service or database.

```
bool_t rpc_gss_get_principal_name(
    rpc_gss_principal_t *principal,
    char *mechanism,
    char *user_name,
    char *node,
    char *secdomain);
```

The type `rpc_gss_principal_t` has been defined earlier.

Principal names are freed using the `free()` library call. A principal name need only be freed in those instances where it was constructed by the application. Values returned by other routines point to structures already existing in a context, and need not be freed.

Returning Errors

The following routine is used to fetch the error code when one of the `rpc_gss_*` routines fails:

```
typedef struct {
    int rpc_gss_error;
    int system_error;
} rpc_gss_error_t;

void rpc_gss_get_error(rpc_gss_error_t
    *error);

/*
 * error code definitions
 */
#define RPC_GSS_ER_SUCCESS      0
#define RPC_GSS_ER_SYSTEMERROR 1
/*
 * more to be defined later
 */
```

If the error is `RPC_GSS_ER_SYSTEMERROR`, the error encountered was a system error. The field `system_error` is then set to one of the possible values for `errno`. Note that unless the invoked function indicates a failure, `rpc_gss_get_error()` will not return a meaningful value.

Miscellaneous Convenience Calls

The following functions are used to obtain information about installed mechanisms and the version of RPCSEC_GSS supported.

```
char **rpc_gss_get_mechanisms();
```

This function fetches the list of supported security mechanisms as a NULL terminated list of character strings.

```
char **rpc_gss_get_mech_info(
    char *mechanism,
    rpc_gss_service_t *service);
```

This function fetches the relevant mechanism information. Supported QOPs are returned as a NULL terminated list of character strings. The returned value for service is one of the values of the `rpc_gss_service_t` type.

```
bool_t rpc_gss_get_versions(
    int *vers_hi,
    int *vers_lo);
```

This returns the highest and lowest RPCSEC_GSS versions supported.

```
bool_t rpc_gss_is_installed(
    char *mechanism);
```

This function returns TRUE if the mechanism is installed, and FALSE otherwise.

Programming Examples

This section provides an indication of how clients and servers will use the interfaces described in this document.

The Client Side

The following is a very simple client program that makes one secure call to a server. Details such as error checking are not shown.

```
CLIENT *clnt; /* client handle */
char server_host[] = "foo";
char service_name[]= "bar@foo.eng.sun.com";
char mech[] = "kerberos_v5";

clnt = clnt_create(server_host, SERV_PROG,
    SERV_VERS, "netpath");
clnt->cl_auth =
    rpc_gss_seccreate(clnt, service_name,
        mech, rpc_gss_svc_integrity,
        NULL, NULL, NULL);
clnt_call(clnt, SERV_PROC1,
    xdr_arg,arg, xdr_res, res, timeout);
AUTH_DESTROY(clnt->cl_auth);
clnt_destroy(clnt);
```

The Server Side

The server should set the name of the principal it is acting as in the main program. Error checking details are not shown.

```
char mech[] = "kerberos_v5";
char service_name[] =
    "bar@foo.eng.sun.com";
int res;
res = rpc_gss_set_svc_name(service_name,
mech, 0,SERV_PROG, SERV_VERS);
```

The following is a simple server side dispatch procedure. Details such as error checking are not shown.

```
static void
server_prog(struct svc_req *rqstp,
    SVCXPRT *xprt)
{
    rpc_gss_ucred_t *ucred;
    rpc_gss_rawcred_t *rcred;

    if (rqstp->rq_proc == NULLPROC) {
        svc_sendreply(xprt, xdr_void, NULL);
        return;
    }
    /*
     * authenticate all other requests
     */
    switch (rqstp->rq_cred.oa_flavor) {
    case RPCSEC_GSS:
      /*
       * get credential information
       */
       rpc_gss_getcred(rqstp, &rcred,
           &ucred, NULL);
      /*
       * verify that user is allowed to access
       * using received security parameters by
       * peeking into my config file
       */
       if (!authenticate_user(ucred->uid,
           rcred->mechanism, rcred->qop,
           rcred->service)) {
              svcerr_weakauth(xprt);
              return;
       }
       break;  /* allow the user in */
    default:
       svcerr_weakauth(xprt);
       return;
    } /* end switch */

    switch (rqstp->rq_proc) {
    case SERV_PROC1:
```

```
    ...
}

    /* usual request processing */
    /* send response */

    return;
}
```

THE RPCSEC_GSS SECURITY PROTOCOL

An RPC session based on the RPCSEC_GSS security flavor consists of three phases: context creation, RPC data exchange, and context destruction. The following section discusses protocol elements for these three phases.

For convenience, **APPENDIX B** of this document reproduces the XDR language description of the RPC protocol. The following description of the protocol uses some of these definitions.

Context creation and destruction use control messages that are not dispatched to service procedures registered by an RPC server. The program and version numbers used in these control messages are the same as the service program and version numbers. The procedure number used is NULLPROC. A field in the credential information (the `gss_proc` field) specifies whether a message is to be interpreted as a control message or a regular RPC message. If this field is set to RPCSEC_GSS_NULL, no control action is implied; in this case, it is a regular data message. If this field is set to any other value, a control action is implied. This behavior is described in the following sections.

The following definitions are used for describing the protocol.

```
/* RPCSEC_GSS control procedures */
#define RPCSEC_GSS_NULL 0
#define RPCSEC_GSS_INIT 1
#define RPCSEC_GSS_CONTINUE_INIT 2
#define RPCSEC_GSS_DESTROY 3
/* RPCSEC_GSS services */
enum rpc_gss_service_t {
    rpc_gss_svc_default = 0,
    rpc_gss_svc_none = 1,
    rpc_gss_svc_integrity = 2,
    rpc_gss_svc_privacy = 3
};
/* Credential */
struct rpc_gss_cred_t {
    unsigned int version;
    unsigned int gss_proc;
    unsigned int seq_num;
    rpc_gss_service_t service;
    opaque handle<>;
};
/* Maximum sequence number value */
#define MAXSEQ 0x80000000
```

Context Creation

Before RPC data is exchanged on a session using the RPCSEC_GSS flavor, a context must be set up between the client and the server. Context creation may involve zero or more RPC exchanges. The number of exchanges depends on the security mechanism. For KerberosV5, context creation involves one RPC exchange.

Context Creation Requests

The first RPC request from the client to the server initiates context creation for those mechanisms that require context creation messages. The program and version used in the RPC message are always those for the service being accessed. The procedure is set to NULLPROC.

The credential field in the RPC message header has the following structure (reproduced from RPC protocol definition):

```
struct opaque_auth {
    sec_flavor flavor;
    opaque body<400>;
};
```

The credential uses the RPCSEC_GSS flavor. The credential body is created by XDR encoding the `rpc_gss_cred_t` structure listed earlier into a byte stream, and then opaquely encoding this byte stream as the body field.

The values of the fields contained in the `rpc_gss_cred_t` structure are set as follows: The version field is set to the current RPCSEC_GSS version (1). The `gss_proc` field must be set to `RPCSEC_GSS_INIT` for the first creation request. In subsequent creation requests, the `gss_proc` field must be set to `RPCSEC_GSS_CONTINUE_INIT`. The server must ignore the `seq_num` field in all creation requests. The server must also ignore the `service` field in all creation requests. In the first creation request, the `handle` field is NULL (opaque data of zero length). In subsequent creation requests, `handle` must be equal to the value returned by the server. `handle` serves as the identifier for the context, and will not change for the duration of the context.

The `opaque_auth` structure also describes the verifier field in the RPC message header. All creation requests have the NULL verifier (AUTH_NONE flavor with zero length opaque data).

The following structure describes the call argument for all creation requests:

```
struct rpc_gss_init_arg {
    opaque gss_token<>;
    unsigned int qop;
    rpc_gss_service_t service;
};
```

Here, `gss_token` is the token returned by the call to `gss_init_sec_context()`, opaquely encoded. The value of this field will probably differ in each creation request, if there is more than one creation request. If the call to `gss_init_sec_context()` does not return a token, the context should have been created (assuming no errors). No more creation requests will occur.

One reason for "overloading" the call data in this way, rather than packing the GSS token into the RPC header, is that RPC Version 2 restricts the amount of data that can be

sent in the header. The opaque body of the credential and verifier fields can be each at most 400 bytes long, and GSS tokens can be longer.

The qop and service fields must be set to the quality-of-protection (QOP) [4] and service the client wishes to use for the session. Some services will need this information before they will allow the context to be set up. Although the client can change the QOP and service used on a per request basis, this may not be acceptable to all RPC-based services. These services may choose to enforce the QOP and service specified during context creation for the rest of the session. If there is more than one creation request, all of them should use the same values for qop and service. The GSS-API interprets a value of 0 (zero) for QOP as a request to use the default QOP level. Since QOP values are mechanism-specific, 0 should be used whenever possible, to be mechanism-independent.

Context Creation Response - Successful Acceptance

The response to a successful creation request is an ACCEPTED response with a status of SUCCESS. The response contains a result argument that has the following structure:

```
struct rpc_gss_init_res {
    opaque handle<>;
    unsigned int gss_major;
    unsigned int gss_minor;
    unsigned int seq_window;
    opaque gss_token<>;
};
```

Here, handle is non-NULL opaque data that serves as the context identifier. The client must use this value in all subsequent requests, whether control messages or otherwise. The gss_major and gss_minor fields contain the results of the call to gss_accept_sec_context() executed by the server. If gss_major is not one of GSS_S_COMPLETE or GSS_S_CONTINUE_NEEDED, the context setup has failed. In this case, the server must set handle and gss_token to NULL. The value of gss_minor depends on the value of gss_major and the security mechanism used. The gss_token field contains any token returned by the gss_accept_sec_context() call executed by the server. A token may be returned for both successful values of gss_major. If the value is GSS_S_COMPLETE, it indicates that the server does not expect any more tokens. If the value is GSS_S_CONTINUE_NEEDED, the server expects another token, and, therefore, at least one more creation request carrying the required token is needed.

In a successful response, the seq_window field is set to the sequence window length supported by the server for this context. This window specifies the maximum number of client requests that may be outstanding for this context. The server will accept seq_window requests at a time. These may be out of order. The client may use this number to determine the number of threads that can simultaneously send requests on this context.

The verifier used for a context creation response is the NULL verifier (AUTH_NONE flavor with zero-length opaque data).

Context Creation Response - Unsuccessful Cases

An ACCEPTED reply to a creation request whose status is other than SUCCESS also has a NULL verifier, and is formulated as usual for different status values.

A DENIED reply to a creation request is also formulated as usual. Two new values, RPCSEC_GSS_NOCRED and RPCSEC_GSS_FAILED, have been defined for the auth_stat type. When the reason for denial of the request is AUTH_ERROR, one of the two new values could be returned in addition to the existing values. These values don't have any special significance for responses to creation requests, but they do for responses to normal (data) requests, as described later.

RPC Data Exchange

The data exchange phase is entered after a context has been successfully set up. The format of the data exchanged depends on the security service used for the request. Although clients can change the security service and QOP used on a per-request basis, this may not be acceptable to all RPC services. For all three modes of service (no data integrity, data integrity, data privacy), the RPC request header has the same format.

RPC Request Header

For the RPC request header, the credential has the opaque_auth structure described earlier. The flavor field is set to RPCSEC_GSS. The credential body is created by XDR encoding the rpc_gss_cred_t structure listed earlier into a byte stream, and then opaquely encoding this byte stream as the body field.

Values of the fields contained in the rpc_gss_cred_t structure are set as follows:

- The version field is set to the current RPCSEC_GSS version (1).
- The gss_proc field is set to RPCSEC_GSS_NULL.
- The service field is set to indicate the desired service (rpc_gss_svc_none, rpc_gss_svc_integrity, or rpc_gss_svc_privacy).
- The handle field is set to the context handle value received from the RPC server during context creation.
- The seq_num field can start at any value below MAXSEQ, and should be incremented (by one or more) for successive requests.

Use of sequence numbers is described in detail later in this paper.

The verifier has the opaque_auth structure described earlier. The flavor field is set to RPCSEC_GSS. The body field is set as follows: The checksum of the RPC header (up to and including the credential) is computed using the gss_sign() call with the desired QOP. gss_sign returns the checksum as an opaque byte stream

and its length. This is encoded into the body field. Note that the QOP is not explicitly specified anywhere in the request. It is implicit in the checksum or encrypted data. The same QOP value used for the header checksum must also be used for the data (for checksumming or encrypting), unless the service used for the request is `rpc_gss_svc_none`.

RPC Request Data - No Data Integrity

If the service specified is `rpc_gss_svc_none`, the data (procedure arguments) are not integrity or privacy protected. They are sent in exactly the same way as they would be if the AUTH_NONE flavor were used (following the verifier). Note, however, that because the RPC header is integrity protected, the sender will still be authenticated in this case.

RPC Request Data - With Data Integrity

When data integrity is used, the request data is represented as follows:

```
struct rpc_gss_integ_data {
    opaque databody_integ<>;
    opaque checksum<>;
};
```

The `databody_integ` field is created as follows: First, a structure consisting of a sequence number, followed by the procedure arguments, is constructed, as shown below as the type `rpc_gss_data_t`:

```
struct rpc_gss_data_t {
    unsigned int seq_num;
    proc_req_arg_t arg;
};
```

Here, `seq_num` must have the same value as the credential. The type `proc_req_arg_t` is the procedure-specific XDR type describing the procedure arguments, thus is not specified here. The byte stream corresponding to the `rpc_gss_data_t` structure and its length is encoded as the `databody_integ` field.

The `checksum` field represents the checksum of the byte stream corresponding to the `rpc_gss_data_t` structure. (Note that this is not the checksum of the `databody_integ` field.) This checksum is obtained using the `gss_sign()` call, with the same QOP as was used to compute the header checksum (in the verifier). The `gss_sign()` call returns the checksum as an opaque byte stream and its length. The checksum and its length are opaquely encoded as the `checksum` field.

RPC Request Data - With Data Privacy

When data privacy is used, the request data is represented as follows:

```
struct rpc_gss_priv_data {
    opaque databody_priv<>
};
```

The `databody_priv` field is created as follows: the `rpc_gss_data_t` structure is reconstructed in the same way as for data integrity. Next, the `gss_seal()` call is invoked to encrypt the byte stream corresponding to the `rpc_gss_data_t` structure, using the same value for QOP as was used for the header checksum (in the verifier). The `gss_seal()` call returns an opaque byte stream (representing the encrypted `rpc_gss_data_t` structure) and its length. These values are used to opaquely encode the RPC request data as the `databody_priv field`.

Server Processing of RPC Data Requests - Context Management

When the server receives a request, the following are verified as acceptable:

- Version number in the credential.
- Service specified in the credential.
- Context handle specified in the credential.
- Sequence number specified in the credential (as explained in this section).
- Header checksum in the verifier.

The `gss_proc` field in the credential must be set to `RPCSEC_GSS_NULL` for data requests. Otherwise, the message will be interpreted as a control message, as discussed later.

The server maintains a window of `seq_window` sequence numbers, starting with the last sequence number seen and extending backwards. If the server receives a sequence number higher than the last number seen, the window moves forward to the new sequence number. If the last sequence number seen is `n`, the server can receive requests with sequence numbers in the range `n` through `(n - seq_window + 1)`, both inclusive. If the sequence number received falls below this range, the server silently discards it. If the sequence number is within this range, and the server has not seen it, the request is accepted. Then, the server turns on a bit to "remember" that it has seen this sequence number. If the server determines that it has already seen a sequence number within the window, it silently discards the request.

The reason for discarding requests silently is because the server cannot determine why the duplicate or out of range request occurred. It could be due to a sequencing problem in the client, network, or operating system; to some quirk in routing; or to a replay attack by an intruder. Discarding the request allows the client to recover after timing out, if indeed the duplication was unintentional or well intended.

The data is decoded according to the service specified in the credential. In the case of integrity or privacy, the server ensures that the QOP value is acceptable, and that it is the same as that used for the header checksum in the verifier. For the same reason, the server will reject the message if the sequence number embedded in the request body is different from the sequence number in the credential. This prevents *splicing* attacks. If the sequence numbers were not compared, an attacker could capture previous RPC requests, and splice in the header of one with the body of another.

Server Reply - Request Accepted

An ACCEPTED reply to a request in the data exchange phase will have the verifier set to the checksum of the sequence number (in network order) of the corresponding request. The QOP used is the same as the QOP for the corresponding request.

If the status of the reply is not SUCCESS, the rest of the message is formatted as usual, as for the AUTH_NULL security flavor.

If the status of the message is SUCCESS, the format of the rest of the message depends on the service specified in the corresponding request message. Basically, what follows the verifier in this case are the procedure results, formatted in different ways depending on the requested service.

If no data integrity was requested, the procedure results are formatted as for the AUTH_NULL security flavor.

If data integrity was requested, the results are encoded exactly as the procedure arguments were in the corresponding request. (Discussed previously in the section under the heading ***RPC Request Data - With Data Integrity.***) The only difference is that the structure representing the procedure's result - `proc_res_arg_t` - must be substituted in place of the request argument structure `proc_req_arg_t`. The QOP used for the checksum must be the same as that used for constructing the reply verifier.

If data privacy was requested, the results are encoded in exactly the same way as the procedure arguments were in the corresponding request. See the section ***RPC Request Data - With Data Privacy.*** The QOP used for encryption must be the same as that used for constructing the reply verifier.

Server Reply - Request Denied

A DENIED reply to a data request is formulated as usual. Two new values, `RPCSEC_GSS_NOCRED` and `RPCSEC_GSS_FAILED`, have been defined for the `auth_stat` type. When AUTH_ERROR is the reason for denial of the request, one of the two new values could be returned in addition to the existing values. These two new values have special significance, differing from the existing reasons for denial of a request.

The server maintains a list of contexts for the clients, with which it is currently in session. Normally, a context is destroyed when the client ends its corresponding session. However, due to resource constraints, the server may destroy a context prematurely (for example, on an LRU basis, or if the server machine is rebooted). In this case, when a client request comes in, there may not be a context corresponding to its handle. The server rejects the request, with the reason RPCSEC_GSS_NOCRED. Upon receiving this error, the client must refresh the context — that is, reestablish it after destroying the old one — and try the request again. This error is also returned if the context handle matches that of a different context that was allocated after the client's context was destroyed. (This will be detected by a failure in verifying the header checksum.)

When the client's sequence number exceeds the maximum the server will allow, the server will reject the request with the reason RPCSEC_GSS_FAILED. Also, if security credentials become stale while in use (for example, ticket expiration in the case of the Kerberos V5 mechanism), the resulting failures cause the RPCSEC_GSS_FAILED reason to be returned. In these cases, the client must refresh the context, and retry the request.

For other errors, retrying will not rectify the problem. The client should not refresh the context until the problem causing request denial is rectified.

Context Destruction

When the client is done using the session, it should send a control message informing the server that it no longer requires the context. This message is formulated just like a data request packet, with the following differences: the credential has `gss_proc` set to RPCSEC_GSS_DESTROY, the procedure specified in the header is NULLPROC, and there are no procedure arguments. For the server to accept the message, the sequence number in the request should be valid, and the header checksum in the verifier should be valid.

The server sends a response as it would for a data request. The client and server should then destroy the context for the session.

If the request to destroy the context fails for some reason, the client need not take any special action. The server should handle situations where clients never inform when they no longer are in session and no longer need the server to maintain a context. The server should use an LRU mechanism or an aging mechanism to clean up in such cases.

COMPARISON TO OPENVISION'S AUTH_GSSAPI

Sun used OpenVision's AUTH_GSSAPI as a prototype for RPCSEC_GSS. In the paper "GSS-API Security for ONC RPC," [7] the goal was to implement a no-frills RPC authentication system with minimal effort.

As B. Jaspan mentioned in this paper, the implementation had the following limitations:

- It did not protect the RPC call message header from modification.
- The AUTH_GSSAPI implementation was not thread-safe.

RPCSEC_GSS solved the first limitation by using the `gss_sign()` primitive to sign the RPC call header (up to and including the credential) and placing the result in the RPC message's verifier.

RPCSEC_GSS solved the second limitation by using the sliding sequence window. This allows a client to have multiple threads making requests on the same client handle. The use of the sequence window also simplified replay

detection on the server side. Note that in the current implementation, only the kernel level client and server code is thread-safe. In the user level version, only the server side is safe because the current user level, client side of ONC RPC is not multithreaded.

The AUTH_GSSAPI implementation did not allow selective use of integrity or encryption on call data; all call data was encrypted. Since RPCSEC_GSS was envisioned for a wide variety of applications (such as NFS and NIS+), always encrypting the call data was considered too restrictive. Also, United States export control laws provided sufficient motivation for an exportable implementation that does not encrypt call data.

This limitation was solved by adding the `service` field to the RPCSEC_GSS credential. This allows the client to change the service requested on each request. Therefore, servers can require that a particular service to be used, and enforce that policy by "locking" the service for a session.

INTEROPERABILITY ISSUES

An RPC client executing in a run time environment without support for RPCSEC_GSS can continue to interoperate with an RPC server that has RPCSEC_GSS support, provided of course the RPC server is not implemented or configured to insist on requests using RPCSEC_GSS. An RPC client that uses RPCSEC_GSS to attempt procedure calls with a server in a non-RPCSEC_GSS environment will not be able to interoperate. An NFS ([11]) implementation is an example of an application that has to cope with these interoperability issues. Version 3 of the NFS protocol ([10]) provides a way to handle this via the security flavor negotiation feature of the corresponding version of the MOUNT protocol.

PERFORMANCE

Extensive performance testing has not been done at this point. Some preliminary tests on a prototype have been run. The results are graphed in Figure 3, with raw data in Table 1. These results were obtained by sending messages of varying length of opaque data (the horizontal axis) from an SS5 to an SS20 over UDP, and taking the average time (the vertical axis, in milliseconds) over 1000 calls. The GSS-API mechanism used with RPCSEC_GSS was Kerberos V5 with mutual authentication and replay detection enabled.

The results match expectations. Using `rpc_gss_svc_none` (call header integrity), as opposed to AUTH_NONE, adds overhead to each message. The overhead is a constant. It is not dependent on the size of the data being sent (because the header size doesn't change). The overhead imposed by `rpc_gss_svc_integrity` and `rpc_gss_svc_privacy` does increase with the size of the data, also as expected. Some profiling work is planned to see if the overhead can been lessened.

Adding data integrity and privacy to ONC RPC is not cheap. There is a trade-off between security and performance. RPCSEC_GSS gives one the ability to enable integrity and privacy of data when needed. Sites that do not need integrity and privacy can still benefit from the strong authentication gained by using RPCSEC_GSS.

data size in bytes	*Average response time (in milliseconds) of Security Flavor*			
	AUTH_NONE	*RPCSEC_GSS services with the Kerberos V5 mechanism*		
		none	*integrity*	*privacy*
256	3.68	5.06	7.20	10.48
512	5.35	6.73	9.80	15.98
1024	8.14	9.58	14.67	26.49
2048	13.70	15.14	23.35	47.10
4096	25.13	26.42	42.19	88.75
8192	47.99	50.20	78.40	170.84

Table 1 Raw Performance Data of RPCSEC_GSS

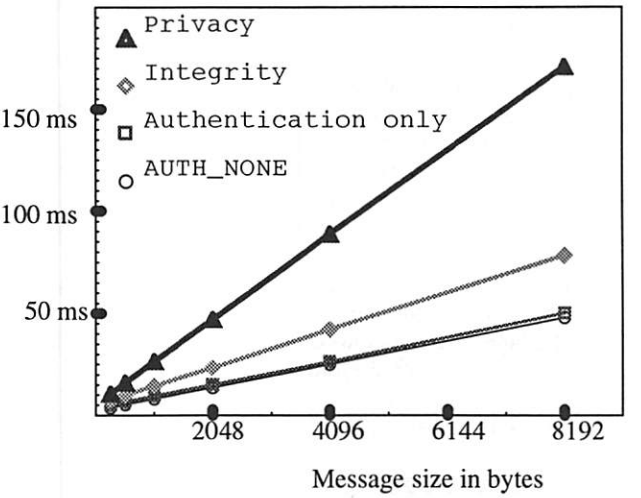


Figure 3 Performance of RPCSEC_GSS

CONCLUSIONS

A working prototype of the user-level and kernel-level versions of RPCSEC_GSS is available. Plans are well underway for adding RPCSEC_GSS support to NFS, to provide a version of NFS with strong authentication, as well as data integrity and privacy.

ACKNOWLEDGMENTS

Much of this work was based on an earlier prototype by OpenVision Technologies. Without their efforts, RPCSEC_GSS may never have happened. In particular, thanks are extended to Barry Jaspan, Mark Horowitz, John

Linn, Ellen McDermott, and the rest of the OpenVision crew at the time the prototype was developed.

On the Sun side, Raj Srinivasan evolved OpenVision's prototype into RPCSEC_GSS, along with help from Mike Eisler. Roland Schemers took Raj Srinivasan's prototype and put the finishing touches on it, along with modifying `snoop` to understand about RPCSEC_GSS.

Dan Nessett and Derek Atkins developed an in-kernel GSS-API library. Lin Ling ported user level RPCSEC_GSS into the kernel and thus achieved the milestone of encrypted NFS traffic on the network.

The authors appreciate the efforts by Rosalind HaLevi, Lin Ling, and Kathy Slattery in reviewing this paper on short notice.

REFERENCES

1. Srinivasan, R. (1995). RFC 1831, "RPC: Remote Procedure Call Protocol Specification Version 2."
2. Srinivasan, R. (1995). RFC 1832, "XDR: External Data Representation Standard."
3. Srinivasan, R. (1995). RFC 1833, "Binding Protocols for ONC RPC Version 2."
4. Linn, J. (1993). RFC 1508, "Generic Security Service Application Program Interface."
5. Wray, J. (1993). RFC 1509, "Generic Security Service API: C-bindings."
6. Kohl, J and Neuman, C. (1993). RFC 1510, "The Kerberos Network Authentication Service (V5)."
7. Jaspan, B. (1995). "GSS-API Security for ONC RPC," *'95 Proceedings of The Internet Society Symposium on Network and Distributed System Security*, pp. 144-151.
8. Neuman, C and Ts'o, T. (1994). "Kerberos: An Authentication System for Computer Networks, *IEEE Communications*, 32.
9. Linn, J. (1996). "The Kerberos Version 5 GSS-API Mechanism," IETF Internet-Draft.
10. B. Callaghan, B. Pawlowski, and P. Staubach, (1995). RFC 1813, "NFS Version 3 Protocol Specification."
11. Sun Microsystems, Inc., (1989). RFC 1094, "NFS: Network File System Protocol specification."

APPENDIX A: SNOOP

The Solaris `snoop` program has been modified so that it understands the RPCSEC_GSS security flavor. The following example uses `snoop` to illustrate the RPCSEC_GSS protocol:

Address List Program

The following RPC Language specification defines an address list server. The server maintains a database of name to address mappings. The following three procedure calls are defined: `addrlist_set`, `addrlist_get`, and `addrlist_del`.

```
const MAX_NAME_LEN = 128;
const MAX_ADDR_LEN = 256;

typedef string name_t<MAX_NAME_LEN>;
typedef string addr_t<MAX_ADDR_LEN>;

struct addr_entry {
    name_t name;
    addr_t address;
};

program ADDRLISTPROG {
  version ADDRLISTVERS {
    bool addrlist_set(addr_entry) = 1;
    addr_entry addrlist_get(name_t) = 2;
    bool addrlist_del(name_t) = 3;
  } = 1;
} = 620756992;
```

Create Context

When the client program calls `rpc_gss_seccreate`, a `RPCSEC_GSS_INIT` control message is sent to the server, and the server sends a reply.

Create Context Request

```
RPC:  ----- SUN RPC Header -----
RPC:
RPC:  Transaction id = 822070204
RPC:  Type = 0 (Call)
RPC:  RPC version = 2
RPC:  Program = 620756992 (GSSTEST), version = 1,
  procedure = 0
RPC:  Credentials: Flavor = 15 (RPCSEC_GSS), len = 20
RPC:     version = 1
RPC:     gss control procedure = 1 (RPCSEC_GSS_INIT)
RPC:     sequence num = 0
RPC:     service = 2 (integrity)
RPC:     handle: length = 0, data = []
RPC:  Verifier   : Flavor = 0 (None), len = 0 bytes
RPC:
RPC:  RPCSEC_GSS_INIT args:
RPC:     gss token: length = 491,data = [491 bytes]
RPC:     quality of protection (qop) = 0
RPC:     service = 2 (integrity)
RPC:
```

Create Context Reply

```
RPC:  ----- SUN RPC Header -----
RPC:
RPC:  Transaction id = 822070204
RPC:  Type = 1 (Reply)
RPC:  This is a reply to frame 1
RPC:  Status = 0 (Accepted)
RPC:  Verifier   : Flavor = 0 (None), len = 0 bytes
RPC:  Accept status = 0 (Success)
RPC:
RPC:  RPCSEC_GSS_INIT result:
RPC:     handle: length = 4, data = [0000000B]
RPC:     gss_major status = 0
RPC:     gss_minor status = 0
RPC:     sequence window  = 128
RPC:     gss token: length = 102, data = [102 bytes]
RPC:
```

Note the value of the handle returned by the server. This value will be used with all future communications with the server.

Data Exchange

Once the context is established, the normal call/reply exchange occurs. The next example shows three exchanges between the client and server. The first uses the integrity service (`rpc_gss_svc_integrity`), the second uses the privacy service (`rpc_gss_svc_privacy`), and the third uses no service (`rpc_gss_svc_none`).

Data Exchange - Integrity (call)

```
RPC:  ----- SUN RPC Header -----
RPC:
RPC:  Transaction id = 822070205
RPC:  Type = 0 (Call)
RPC:  RPC version = 2
RPC:  Program = 620756992 (GSSTEST), version = 1,
    procedure = 1
RPC:  Credentials: Flavor = 15 (RPCSEC_GSS), len = 24
RPC:     version = 1
RPC:     gss control procedure = 0 (RPCSEC_GSS_NULL)
RPC:     sequence num = 2
RPC:     service = 2 (integrity)
RPC:     handle: length = 4, data = [0000000B]
RPC:  Verifier    : Flavor = 15 (RPCSEC_GSS), len = 33
RPC:  [601F06052B0501050201010000...4170507FD073EE6BAC1]
RPC:
RPC:  RPCSEC_GSS data seq_num = 2
RPC:
GSS:  ----- Sun GSS TEST -----
GSS:
GSS:  Proc = 1 (Set address value)
GSS:  name  = schemers
GSS:  value = roland.schemers@eng.sun.com
GSS:
RPC:  ----- RPCSEC_GSS -----
RPC:
RPC:  checksum: len = 33
RPC:  [601F06052B0501050201010...440D6D951CF8126DE80C3]
RPC:
```

Note that since integrity (and not privacy) was used, call arguments are visible to `snoop`. However, since they are integrity protected, an attacker is unable to modify them. Also note that the sequence number (2 in this case) is the same in both the call header and body. As mentioned earlier, this prevents a *splicing* attack.

Data Exchange - Integrity (reply)

```
RPC:  ----- SUN RPC Header -----
RPC:
RPC:  Transaction id = 822070205
RPC:  Type = 1 (Reply)
RPC:  This is a reply to frame 3
RPC:  Status = 0 (Accepted)
RPC:  Verifier    : Flavor = 15 (RPCSEC_GSS), len = 33
RPC:  [601F06052B0501050201010000...8099DD1E993391D5F]
RPC:  Accept status = 0 (Success)
RPC:
RPC:  RPCSEC_GSS data seq_num = 2
RPC:
GSS:  ----- Sun GSS TEST -----
GSS:
GSS:  Proc = 1 (Set address value)
GSS:  result = True
GSS:
RPC:  ----- RPCSEC_GSS -----
RPC:
RPC:  checksum: len = 33
RPC:  [601F06052B0501001010000...CBB5F757967F47307610]
RPC:
```

Note that the reply arguments are also integrity protected.

Data Exchange - Privacy (call)

```
RPC:  ----- SUN RPC Header -----
RPC:
RPC:  Transaction id = 822070206
RPC:  Type = 0 (Call)
RPC:  RPC version = 2
RPC:  Program = 620756992 (GSSTEST), version = 1,
    procedure = 2
RPC:  Credentials: Flavor = 15 (RPCSEC_GSS), len = 24
RPC:     version = 1
RPC:     gss control procedure = 0 (RPCSEC_GSS_NULL)
RPC:     sequence num = 3
RPC:     service = 3 (privacy)
RPC:     handle: length = 4, data = [0000000B]
RPC:  Verifier    : Flavor = 15 (RPCSEC_GSS), len = 33
bytes
RPC:  [601F06052B0501050201010000...D641560DD62123393]
RPC:
RPC:  RPCSEC_GSS (CALL args encrypted)
RPC:
```

Data Exchange - Privacy (reply)

```
RPC:  ----- SUN RPC Header -----
RPC:
RPC:  Transaction id = 822070206
RPC:  Type = 1 (Reply)
RPC:  This is a reply to frame 5
RPC:  Status = 0 (Accepted)
RPC:  Verifier    : Flavor = 15 (RPCSEC_GSS), len = 33
RPC:  [601F06052B0501050201010000...F624B44EAE9008C492E]
RPC:  Accept status = 0 (Success)
RPC:
RPC:  RPCSEC_GSS (REPLY args encrypted)
RPC:
```

Since the call and reply arguments are encrypted, `snoop` can't display them. It could display the encrypted data as hexadecimal. But, without knowing the encryption key display, the hexadecimal data is not very useful.

Data Exchange - Service None (call)

```
RPC:  ----- SUN RPC Header -----
RPC:
RPC:  Transaction id = 822070207
RPC:  Type = 0 (Call)
RPC:  RPC version = 2
RPC:  Program = 620756992 (GSSTEST), version = 1,
    procedure = 3
RPC:  Credentials: Flavor = 15 (RPCSEC_GSS), len = 24
RPC:     version = 1
RPC:     gss control procedure = 0 (RPCSEC_GSS_NULL)
RPC:     sequence num = 4
RPC:     service = 1 (none)
RPC:     handle: length = 4, data = [0000000B]
RPC:  Verifier    : Flavor = 15 (RPCSEC_GSS), len = 33
RPC:  [601F06052B0501050201010000...FF1005156F24570E20A]
RPC:
GSS:  ----- Sun GSS TEST -----
GSS:
GSS:  Proc = 3 (Delete address value)
GSS:  name  = schemers
GSS:
```

Data Exchange - Service None (reply)

```
RPC:  ----- SUN RPC Header -----
RPC:
RPC:  Transaction id = 822070207
RPC:  Type = 1 (Reply)
RPC:  This is a reply to frame 7
RPC:  Status = 0 (Accepted)
RPC:  Verifier    : Flavor = 15 (RPCSEC_GSS), len = 33
bytes
RPC:  [601F06052B0501050201010000...C6409E5707D34AB]
RPC:  Accept status = 0 (Success)
RPC:
GSS:  ----- Sun GSS TEST -----
GSS:
GSS:  Proc = 3 (Delete address value)
GSS:  result = True
GSS:
```

When `rpc_gss_svc_none` is used, the call/reply arguments are not integrity protected or encrypted. The call/reply header is still integrity protected though, providing strong authentication.

Context Destruction

The context will be destroyed when the client calls the standard ONC RPC `auth_destroy` function.

Context Destruction - Request

```
RPC:  ----- SUN RPC Header -----
```

```
RPC:
RPC:  Transaction id = 822070208
RPC:  Type = 0 (Call)
RPC:  RPC version = 2
RPC:  Program = 620756992 (GSSTEST), version = 1,
    procedure = 0
RPC:  Credentials: Flavor = 15 (RPCSEC_GSS), len = 24
RPC:     version = 1
RPC:     gss control procedure = 3 (RPCSEC_GSS_DESTROY)
RPC:     sequence num = 5
RPC:     service = 1 (none)
RPC:     handle: length = 4, data = [0000000B]
RPC:  Verifier    : Flavor = 15 (RPCSEC_GSS), len = 33
RPC:  [601F06052B0501050201010000...A065CC80B17751779]
RPC:
```

Context Destruction - Reply

```
RPC:  ----- SUN RPC Header -----
RPC:
RPC:  Transaction id = 822070208
RPC:  Type = 1 (Reply)
RPC:  This is a reply to frame 9
RPC:  Status = 0 (Accepted)
RPC:  Verifier    : Flavor = 15 (RPCSEC_GSS), len = 33
RPC:  [601F06052B0501050201010000...10A97BCF979D24DFEA]
RPC:  Accept status = 0 (Success)
RPC:
```

APPENDIX B: THE RPC MESSAGE PROTOCOL

For convenience, the XDR language description of the RPC message protocol is reproduced here. For explanations of these constructs, refer to RFC-1831, "RPC Remote Procedure Call Protocol Specification Version 2."

```
/* RPC message type */
enum msg_type {
    CALL  = 0,
    REPLY = 1
};
/* Reply types */
enum reply_stat {
    MSG_ACCEPTED = 0,
    MSG_DENIED   = 1
};
/* Security flavors */
enum sec_flavor {
    AUTH_NONE = 0,
    AUTH_SYS = 1,
    AUTH_SHORT = 2,
    AUTH_DH = 3,
    AUTH_KERB = 4,
    RPCSEC_GSS = 6
};

/* Status of accepted messages */
enum accept_stat {
    SUCCESS = 0,
    PROG_UNAVAIL = 1,
    PROG_MISMATCH = 2,
    PROC_UNAVAIL = 3,
    GARBAGE_ARGS = 4,
    SYSTEM_ERR = 5
};
/* Status of rejected messages */
enum reject_stat {
    RPC_MISMATCH = 0,
    AUTH_ERROR = 1
};
/* Why authentication failed */
enum auth_stat {
    AUTH_OK = 0,
    /* failed at remote end */
    AUTH_BADCRED = 1,
    AUTH_REJECTEDCRED = 2,
    AUTH_BADVERF = 3,
    AUTH_REJECTEDVERF = 4,
    AUTH_TOOWEAK = 5,
    /* failed locally */
    AUTH_INVALIDRESP = 6,
    AUTH_FAILED = 7,
    /* kerberos v4 errors */
    AUTH_KERB_GENERIC = 8,
    AUTH_TIMEEXPIRE = 9,
    AUTH_TKT_FILE = 10,
    AUTH_DECODE = 11,
    AUTH_NET_ADDR = 12,
    /* RPCSEC_GSS errors */
    RPCSEC_GSS_NOCRED = 13,
    RPCSEC_GSS_FAILED = 14
};
/* Opaque structure of
       credential and verifier */
struct opaque_auth {
    sec_flavor flavor;
    opaque body<400>;
};
/* The RPC message */
struct rpc_msg {
    unsigned int xid;
    union switch (msg_type mtype) {
    case CALL:
        call_body cbody;
    case REPLY:
        reply_body rbody;
    } body;
};
/* Body of RPC call */
struct call_body {
   unsigned int rpcvers;
   unsigned int prog;
   unsigned int vers;
   unsigned int proc;
   opaque_auth cred;
   opaque_auth verf;
   /* procedure specific
      parameters start here */
};
/* Body of RPC reply */
union reply_body switch (reply_stat stat) {
case MSG_ACCEPTED:
    accepted_reply areply;
case MSG_DENIED:
    rejected_reply rreply;
} reply;
/* Accepted reply */
struct accepted_reply {
    opaque_auth verf;
    union switch (accept_stat stat) {
    case SUCCESS:
        opaque results[0];
        /* procedure-specific
                   results start here */
    case PROG_MISMATCH:
        struct {
            unsigned int low;
            unsigned int high;
        } mismatch_info;
    default:
       /*
        * Void. Cases include PROG_UNAVAIL,
```

```
        * PROC_UNAVAIL, GARBAGE_ARGS, and
        * SYSTEM_ERR.
        */
       void;
    } reply_data;
};
/* Rejected reply */
union rejected_reply switch(reject_stat stat)
{
case RPC_MISMATCH:
    struct {
        unsigned int low;
        unsigned int high;
    } mismatch_info;
case AUTH_ERROR:
    auth_stat stat;
};
```

Establishing Identity Without Certification Authorities

Carl M. Ellison*
CyberCash, Inc.

Introduction

> Without a KMI[1] of trusted certificate authorities, users cannot know with whom they are dealing on the network....[3]

It is commonly assumed that if one wants to be sure a public key belongs to the person he hopes it does, he must use an identity certificate issued by a trusted Certification Authority (CA). The thesis of this paper is that a traditional identity certificate is neither necessary nor sufficient for this purpose. It is especially useless if the two parties concerned did not have the foresight to obtain such certificates before desiring to open a secure channel.

There are many methods for establishing identity without using certificates from trusted certification authorities. The relationship between verifier and subject guides the choice of method. Many of these relationships have easy, straight-forward methods for binding a public key to an identity, using a broadcast channel or 1:1 meetings, but one relationship makes it especially difficult. That relationship is one with an old friend with whom you had lost touch but who appears now to be available on the net. You make contact and share a few exchanges which suggest to you that this is, indeed, your old friend. Then you want to form a secure channel in order to carry on a more extensive conversation in private. This case is subject to the man-in-the-middle attack. For this case, a protocol is presented which binds a pair of identities to a pair of public keys without using any certificates issued by a trusted CA.

The apparent direct conflict between conventional wisdom and the thesis of this paper lies in the definition of the word "identity" – a word which is commonly left undefined in discussions of certification.

*cme@cybercash.com

[1]Key Management Infrastructure – a hierarchy of Certification Authorities

Identity

To some people, an "identity" is just a name. To others, it is the data needed to allow one to track down, arrest and punish the violator of a contract or the passer of a bad check. In the case of an old friend who shows up on the net, "identity" closely follows the dictionary definition:

> Identity: 2a. the distinguishing character or personality of an individual.[7]

To the person verifying the identity of an old friend, that other person's identity is a body of memories: an internal representation of the distinguishing character or personality of an entity, as the verifier has come to know that entity through the relationship between them. That body of memories is labeled by a name but the name is not an identity. It is a label for one. It is also neither unique nor global[2].

Each human being is forced to name people in order to think about them. However, people typically use short internal names, of the form: "IBM", "CyberCash", "Rocky", "Alan", "Zorak", "Lolly", "Red", etc. To the person using such a name, the name is unique. To that person, no other qualification is required. However, only the first two come even close to being likely to specify the same entity to everyone else. Even in those cases there is no guarantee. Unlikely though this might be, there *could* be an "Idaho Boiler Mechanics, Inc." of special importance to someone. It is also possible for someone to change his or her name completely – perhaps to start a new life and discard psychological baggage. The identity of that person does not change but his or her generally recognized name does while someone's internal name for that person may or may not change accordingly.

Identity Certificates

The word "certificate" has been used so long to mean "identity certificate" that many people in the field of

[2]One recent attempt to deal with this fact of life can be found in the SDSI[4] proposal of Rivest and Lampson.

cryptography find it difficult to consider any other kind. For the purpose of this discussion, let us start with two definitions:

Definition 1 (Certificate) *A certificate is a digitally signed, structured message which delegates an attribute of some form to a public key.*

Definition 2 (Identity Certificate) *An identity certificate is a certificate which binds the name of an entity to a public key. Its putative meaning is to delegate all the attributes of the named entity to the public key.*

The attributes most commonly considered are **trust** and **authority**. Those words, too, are frequently used without definition when they should be defined, but for the purpose of this paper, each definition applies equally.

There are two commonly used examples of identity certificate at this time: X.509 and PGP.

X.509

X.509 certificates grew out of the X.500 global database design. Under X.500 all entities in the world (not just people) would have unique names, called Distinguished Names (DNs). These names are organized in a hierarchy, so that the name space can be distributed both for storage and for management.

Given a global, unique name for every entity on the planet, when one wanted to bind a public key to an entity, what better than use that unique name? X.509 was the result.

There are some problems with X.509. Two are relevant to this paper. The first is that X.500 has never been deployed worldwide and is likely never to be. The second is that even if some large corporation were to deploy X.500 internally and generate X.509 certificates using those DNs[3], it would make those DNs unique by addition of information of relevance to the corporation, such as operational unit, building name or mail stop. Such a unique name is not adequate to disambiguate a common name from the point of view of all possible users of the certificate. That is, Alice may have an old friend, Bob Jones, at IBM but have no idea of his mail stop, building name or operational unit. Although she has a fully trusted identity certificate from IBM for each employed Bob Jones, she can not trust any of them to be her friend. Therefore, the certificate has failed in its task of binding identity to a public key.

[3]Some organizations may build an X.500 name space solely for the purpose of creating X.509 certificates rather than for X.500's original purpose of providing a global directory service.

PGP

PGP, actively shunning the rigid hierarchy of X.509, permits each user to "certify" a binding between a public key and a UserID. By convention, the UserID consists of a common name and an e-mail address. The e-mail address serves to make the UserID unique, just as the X.509 DN is made unique.

The PGP UserID has one advantage over an X.509 DN, in that an e-mail address is an element of a currently deployed global name space. However, PGP key signatures have a disadvantage compared to X.509 certificates in that the person signing the key is not necessarily either known or trusted. PGP attempts to compensate for that by allowing multiple, presumably independent signatures to vote a binding into validity.

Other Certificates

There are some certificates which are sometimes considered to be identity certificates, possibly because the word "certificate" has traditionally been taken to imply an identity binding.

Driver's license analogy

It is common practice for a merchant accepting a paper check to ask for a driver's license for identification and to write the license number on the check. A driver's license is a physical certificate: an instrument, issued by a large, trusted organization, which binds a person's picture and physical description to his signature. That binding is relatively loose, however. The Maryland driver's license, for example, uses a digitized signature written on a data tablet which is difficult to compare to a normally written signature.

In the digital world, this would be a certificate which binds an image, birth date, height, weight and gender to a signature key. Such a certificate would be of value not only for giving meaningful evidence of identity but also for on-line dating services. It would have to be issued by a trusted third party – one trusted not to lie to the verifier of the certificate.

There is another binding provided by a driver's license. The Department of Motor Vehicles is sure to keep track of licensed drivers. In the event that a check is returned because of insufficient funds and the person passing it fails to respond to appeals to rectify the situation, the merchant can hope to employ the DMV's database to track down the person and institute legal action against him or her.

This kind of certificate also needs to be issued by a trusted third party. Specifically, that party needs

to attest to the ability to locate a person in the event of criminal behavior.

Neither of these certificates is properly an identity certificate[4]. Neither needs to include a person's true name, for example. A certificate with a picture and data about a person could be issued by a dating service and have only a nickname and 1-900 telephone number to identify the person. A certificate attesting to the ability to find a person in case of misdeed could have only a serial number, to be used by the tracking service after it has been paid its fee.

Certificates for electronic commerce

One can perform electronic commerce without identity certificates at all. Certificates are needed but they are needed only to grant authority to a signature key to spend money from a given account. For example, the SET[5] certificate binds an account number to a signature key. Any card-holder name which might also be in the certificate is of no importance in the process of deciding whether to honor a given purchase request.

Specifically, if someone accepts a paper check, that check has a printed name and address as well as a signature and amount. The person depositing the check does not care about either the name or signature. He cares only about whether his bank account is augmented by the amount of the check. The person's bank (the *accepting bank*) cares only about whether the *issuing bank* transfers funds in return for the check. Both care that there is a signature, but neither can verify it. The issuing bank checks the signature and checks that there is money in the indicated account but it, too, ignores the name and address printed on the check.

In cyberspace, the same applies – with the exception that the person accepting an electronic check is able to determine whether the signer of the check is authorized by the issuing bank – given the proper certificates. There is still some trust involved that the indicated account has funds, but there is always that issue unless one has on-line clearing. To determine validity of an electronic check, one needs to verify:

- the digital signature on the check
- a certificate from the issuing bank giving check-writing authority to the signature key
- a certificate from the accepting bank, acknowledging the issuing bank's signature key as belonging to a bank with which it deals (or more likely a chain of certificates through various clearing houses)
- a certificate issued by the person accepting the check (the ultimate authority) verifying the accepting bank's key. That certificate would be generated when the person opened his electronic checking account.

In none of those certificates does a name or other generally recognized mark of identity appear – although each party may file the certificate under a local name.

Local and global names

The internal name each human is forced to use in order to think about a person can be thought of as a local name, as opposed to a global name in the form of an X.500 DN or a fully qualified Internet e-mail address. Even the DN in an X.509 certificate can be thought of as a name local to the CA which issued the certificate.

Local names – nicknames – are both unique and meaningful, but only to the one owner of the local name space.

For a (name,key) certificate actually to bind a key to an identity, there needs to be a secure mapping from the name space of the issuer to that of the verifier. Since the issuer and verifier are free to change their name spaces at will, the two should be linked for periodic update. The simplest way to achieve that linked mapping is for the issuer and the verifier to be the same person. If the verifier makes his own certificates, he can use his own choice of names in those certificates. However, that requires him to assure himself of the validity of the binding without relying on certificates from a third party.

The preferred means for performing that verification depends on the relationship between the subject and the verifier.

Relationships

A verifier can issue his own certificate for someone's public key, binding the key to the verifier's name for that person, in one of a number of ways, depending on the relationship between the two parties: personal contact; major corporation; network acquaintance; stranger; or old friend, out of touch.

[4]The picture certificate could be of use in binding identity of an old friend unless that friend is old enough to have become unrecognizable.

[5]Secure Electronic Transactions – a standard for electronic commerce being developed jointly by VISA and MasterCard

Throughout this section, it must be kept in mind that the certificates being issued are for the verifier's use only. The names involved are the verifier's own nicknames assigned to the corresponding entities. This certificate structure can be extended through SDSI[4] to affect other people, but under no circumstances are the certificates mentioned here pretending to deal in universal, global names. To the contrary, it is clear that there is no such thing as a universal, global name space with names meaningful to all possible users and there never will be. There are too many names for one human being to remember and attach meaning to each one of them.

Personal contact

If the two parties are in personal contact periodically, they can exchange their public keys in person (or alternatively, they can exchange secure hashes of their keys). There are three variants of this exchange, however, and the certificate verifier needs to keep them in mind and note which form of exchange was employed for a given certificate:

- **Alice gives her key to Bob**

  Bob is sure Alice claims to own the key he is about to build into a certificate. As long as that key is used only for confidentiality, Alice would gain nothing by lying about ownership of the key. Similarly, if the key is for signatures but the only thing being signed is a key exchange for confidentiality, Alice would have no reason to lie.

  However, if the key were used to sign lab notebooks, later to be used in patent applications, for example, Alice might have a reason to claim ownership of a key she can not actually use.

- **Alice gives her key to Bob; Bob gives a challenge to Alice**

  This variant can take place over an extended period of time. During the personal meeting, Bob gives a challenge value to Alice to take home and process. Later, Bob receives an electronic communication from Alice completing her half of the protocol. Bob gains all the assurance in this case which he received in the previous but holds off generating a certificate until Alice responds correctly to the challenge.

  If Alice's is a signature key, Alice demonstrates the ability to sign the random challenge which had been given to her in the clear. If Alice's is a confidentiality key, Bob hands Alice a message with the random challenge encrypted under her key and generates a certificate only after Alice returns that challenge value.

  In this variant, Bob is assured that Alice has access to the private key, although that access might be by duping the true owner of the key into acting as an oracle for Alice.

- **Alice gives her key to Bob; Bob witnesses Alice's processing of his challenge**

  In this case, Bob is assured of everything the previous variant assured him and also knows that Alice is in true possession of the private key. For the truly paranoid, Alice can be sent into a sealed room to perform the signature or decryption of the challenge.

These personal exchanges are appropriate not only for employee/employer relationships where the employer might run a traditional CA but also for individual acquaintances and relatives. The personal meeting can be in the flesh or it could be by telephone or video conference, depending on the extent to which the verifier trusts those connections.

Major corporation

A major corporation or other entity with access to broadcast media (radio, TV, satellite, newspaper, magazine) which is difficult or impossible to intercept and modify, can publish its key in an advertisement, binding it to the corporate name. A verifier can trust this binding to a certain extent on the assumption that the corporation will perform a random sampling of the broadcast and detect any tampering. This has little value however if the verifier is being targeted specifically for spoofing[6].

For this relationship, a traditional certificate hierarchy provides a benefit. However, if a verifier is being targeted, the attacker can presumably substitute a root certificate and certificate chain. Anyone wanting to provide substantial assurance via certificate hierarchies in this case needs to assure the delivery of the correct root key to every individual verifier.

Fortezza/Tessera provides that assurance by storing the root key at time of card programming and prohibiting later modification of that key.

A less heavily structured option is to gather some keys for yourself in person, by visiting the companies in question, and then passing your knowledge around to your friends, in the manner of the PGP web of trust, assuming of course that SDSI[4] naming is employed.

[6] Even a radio or TV broadcast can not be trusted totally, as the old "Mission Impossible" television series was fond of pointing out.

Network acquaintance

Perhaps the easiest person for whom one can generate a certificate is a network acquaintance. Alice has never met Bob in person and likely never will. She got to know him via the network – exchanging e-mail or, someday, digital voice or video.

Provided all of these exchanges were digitally signed in the same key (or encrypted under the same confidentiality key), Alice can immediately generate a certificate tying her name for Bob to that key. In fact, she can generate that certificate the first time she encounters the key.

This requires no mathematics. It relies on the definition of "identity".

A public key is a surrogate presence in cyberspace for some entity in physical space. It acts directly in cyberspace, just as the associated entity can act in physical space.

The public key is bound tightly to the physical space entity, assuming that the latter maintains physical possession of the associated private key and sole knowledge of the pass phrase under which it is encrypted. Once there are devices which actively monitor living contact with the proper thumb, for example, instead of merely accepting a pass phrase or PIN, the bond between key and person will be even stronger.

If the bond between key and person is broken, no layer of certificates will strengthen it. On the contrary, in this case certificates merely provide a false sense of security to the verifier.

Alice can therefore generate an immediate local-name certificate for this person. She can not testify to that person's global name. She can not know even if he's a dog, much less what gender. However, to the extent that she knows a person, she knows the cyberspace surrogate for that person and in cyberspace, the person's key is the final authority.

This comes back to the word "identity". Alice's total knowledge of Bob – and therefore Bob's total identity, from Alice's point of view – is derived from digital communications strongly tied to a public key. That key is therefore tied more strongly to this person's identity (personality; operation of his mind) than would be a key presented by someone in the flesh.

Alice can issue her certificate for Bob even before reading Bob's first signed message. She can generate that certificate and choose her own nickname for this new person at will. The real person is yet to reveal himself but even with no information, everything Alice knows about Bob (namely an as-yet unread message) is digitally signed in the same key and Bob's total personality is still bound to that key, from Alice's point of view.

Stranger

A stranger provides perhaps the biggest challenge to the thesis that no global identity certificates are needed. There is not even any shared knowledge with which to perform one of the protocols given below and build a local identity certificate.

On the other hand, as the network acquaintance example showed, this might be the easiest case. If someone is a total stranger, then there is no presumed relationship and therefore no trust and therefore nothing to risk by binding his key to a still-meaningless nickname. All trust of a network acquaintance will be acquired over time through digitally signed interactions so the verifier has assurance that all these interactions came from the same person.

Let us differentiate the stranger case from the network acquaintance artificially. Let us assume that we need to extend some trust to this stranger without building a relationship first. The stranger might be a shopper, wanting to spend digitally signed checks on our web site. The stranger might want to gain FTP or telnet access to an ISP's computer. The stranger might want physical access to an electronically controlled door.

In all of these cases, what is needed by the verifier is not an identity certificate. The person being certified is by definition a stranger and therefore his or her identity is meaningless to the verifier. Rather, what the verifier needs is an **authorization certificate**: a certificate which authorizes the holder of a key (therefore, authorizes the key) to spend money from a given bank or charge account, to get access to a given FTP server, etc. These certificates need to be issued by an authority with the power to delegate the authorization in question – a bank, a corporation owning the FTP server, etc. However, these are not identity certificates and this paper will leave the issue of authorization certificates to be addressed elsewhere[2].

Old friend, out of touch

Various protocols are presented later in this paper for using shared knowledge between the two parties involved mutually to verify each other's identity and bind those identities to their respective public keys. Because these depend on shared, common knowledge, each person is assured of binding a key to an identity by the dictionary definition given earlier – that is, to a distinguishing character or personality.

These protocols are complicated by the need to prevent various attacks, some due to the low entropy of individual pieces of shared common knowledge. This is especially a concern because the case of an old friend who shows up on the net is tailor-made for the "man in the middle" attack.

Man In The Middle Attack

The Man In The Middle attack assumes that there are two people, Alice and Bob, who hope to establish a secure communication channel. Sitting between them and controlling the only physical communications channel connecting them is an active eavesdropper, Mallet, who is capable not only of intercepting all traffic but also of deleting, inserting and modifying messages en route.

Alice and Bob try to establish a secure channel by sending an RSA public key to each other over this insecure channel or by engaging in an exponential key exchange protocol such as Diffie-Hellman.

RSA

Using RSA, Alice and Bob each generate an RSA key pair and each transmit their public key to the other. If there is no active eavesdropper, then all other eavesdroppers are foiled.

1. A→: (Alice transmits) K_A, Alice's public key
2. B→: K_B, Bob's public key
3. →B: (Bob receives) K_A
4. →A: K_B

At this point, Alice and Bob have each other's public keys and can use them for a normal secure mail interchange. Alternatively, one of them can generate a key for a symmetric link-encrypting cryptosystem and transmit it to the other, after which they can use that key and have an open secure channel.

With the active eavesdropper, Mallet, in the channel, the situation changes. Mallet generates two RSA key pairs, with public keys K_{MA} and K_{MB}. Alice and Bob engage in the same protocol listed above, or so they think, but in fact:

1. A→: K_A
2. B→: K_B
3. →M: K_A
4. →M: K_B
5. M→B: (Mallet transmits to Bob) K_{MA}
6. M→A: K_{MB}
7. →B: K_{MA}
8. →A: K_{MB}

Now, Alice and Bob believe that they have each other's RSA keys, but instead they have Mallet's versions of each other's key. For all future traffic, whether plaintext or a symmetric key, Mallet will intercept each message, decrypt it, possibly modify it and re-encrypt it in the appropriate destination key. Alice and Bob are unaware of this interference, except possibly for performance anomalies.

In other words, Mallet has created two secure channels, one between himself and Alice over which he impersonates Bob, and the other between himself an Bob over which he impersonates Alice. This general attack applies no matter what public key exchange mechanism is employed.

Diffie-Hellman

Using Diffie-Hellman, Alice and Bob determine a new secret quantity, allegedly known only to the two of them. In the normal protocol:

1. Alice and Bob agree to a large prime, p and a generator g
2. Alice generates a random, secret quantity, x_A
3. Bob generates a random, secret quantity, x_B
4. Alice computes $y_A = g^{x_A} mod p$
5. Bob computes $y_B = g^{x_B} mod p$
6. A→: y_A
7. B→: y_B
8. →B: y_A
9. →A: y_B
10. Alice computes $z = y_B^{x_A} mod p$
11. Bob computes $z = y_A^{x_B} mod p$

With Mallet in control of the raw channel, however, he gains access by forming two independent secure channels – one between himself and Alice and the other between himself and Bob. He then translates all messages between them, modifying some as necessary. For example, Alice and Bob may try to verify security of the channel by transmitting the resulting z or a secure hash of it to each other. Mallet

would need to modify the content of such messages, in order to continue his deception[7]

Protocol for Binding Identity to Keys

Let there be Alice and Bob, old friends who want to communicate securely. They have only one channel between them and it is possible for Mallet to intercept and modify all messages between Alice and Bob. Alice and Bob can perform the following protocol in order to determine whether Mallet has captured their secure channel. If they determine that he has, then they record the shared secret knowledge which they had revealed to Mallet and mark it as suspect. If they determine that the channel is truly secure, then the shared secret knowledge which they used to verify each other's identity and the security of the channel can be reused in the future – and additional shared knowledge can pass between them for possible future use.

1. Alice generates a key pair and sends the public key, K_A, to Bob.
2. Bob generates a key pair and sends the public key, K_B, to Alice.
   (a) If Mallet has captured the channel, he generates keys and sends his keys on to Alice and Bob – sending K_a instead of K_A to Bob, K_b instead of K_B to Alice. In this case, Alice has keys (K_A, K_b) while Bob has keys (K_a, K_B). The allegedly secure channel they create with these keys includes Mallet, able to read all traffic and also to modify it.
   (b) If Mallet is not present in the channel, then Alice and Bob both have keys (K_A, K_B) and the secure channel they create with those keys is private to them.
3. Alice and Bob engage in one or more rounds of protocol. Each round returns either "failure" or an entropy value, E. Further rounds are executed until there are K failures or the sum of returned E values exceeds a security parameter S.
4. If K rounds failed, the protocol failed. Alice and Bob conclude that the channel may have been compromised by Mallet.
5. If $\sum E > S$, the protocol succeeded. Alice and Bob are assured that the probability that Mallet has captured the channel is less than 2^{-S}.

Once the channel has been verified secure with Alice and Bob having proved to their own satisfaction that the other end of the channel holds their old friend, a side effect of the protocol is that Alice and Bob can issue personal identity certificates for their old friend's key – either the key used to create the channel or a signature key exchanged over that secure channel.

Protocol round

For each round of the protocol, Alice and Bob use the presumably secure channel they have set up with the keys they exchanged. They could be correct in assuming that channel to be secure and private to the two of them or there could be an active eavesdropper, Mallet, who has created two secure channels – one between himself and Alice and one between himself and Bob. Each protocol round presents a test to the person on the other end of the immediate secure channel – either the desired party or Mallet. The protocol uses interlock in order to attempt to insure that only the person on the end of the immediate secure channel can provide answers. Alice therefore tests either Mallet or Bob to determine which one he is.

Over the secure channel:

1. Alice and Bob each formulate and transmit a question which only the other should be able to answer, such as, "what was the first name of the woman you kept telling me about in June, 1979?" R_A and R_B are Alice's and Bob's answers, respectively, to each other's questions.
2. Alice and Bob each list the possible answers the other might give, accounting for spelling variations. Alice forms the list $X_i, 0 < i \leq I$, of answers she would accept from Bob and Bob forms the list $Y_j, 0 < j \leq J$, of answers he would accept from Alice. (Each has presumably chosen a question designed to minimize the number of acceptable answers.)
3. Alice computes $U_i = H(X_i|K_A|K_1)$ and $T_A = H(R_A|K_1|K_A)$ where H() is a cryptographically strong hash and K_1 is the key Alice believes belongs to Bob.
4. Bob computes $V_j = H(Y_j|K_B|K_2)$ and $T_B = H(R_B|K_2|K_B)$, where K_2 is the key Bob believes belongs to Alice.

[7] A secure telephone can present problems to Mallet in this desire. He would be forced to be able to imitate the voices of the participants if he wanted to change any messages. Assuming the parties involved knew each other's voices, this could be significantly difficult.

If there is no eavesdropper and if $R_B \in \{X_i\}$ and if $R_A \in \{Y_j\}$, then $T_B \in \{U_i\}$ and $T_A \in \{V_j\}$.

If there is an eavesdropper who has replaced either K_A or K_B then the probability that $T_A \in \{V_j\}$ is roughly $J/2^S$, where $S = \min(|H|, |K|)$. Since $|K| > |H|$ in common practice, this probability becomes $J2^{-|H|}$, for large $|H|$ and small J, reflecting the assumption that one is unable to find a collision for a cryptographically strong hash function with probability greater than $2^{-|H|}$.

5. Alice and Bob exchange answers T_A and T_B, using an interlock protocol detailed below. If the received $T'_A \in \{V_j\}$ and $T'_B \in \{U_i\}$, then the round is considered successful. Otherwise, the round has failed. (Failure of the round might be detected in the middle of the interlock protocol, in which case the interlock exchange is terminated early, depriving Mallet of information.)

6. If the round succeeds, Alice computes the entropy, $E_{A,i}$ of answers X_i, given the transmitted question, and computes

$$P_A = \sum_{i=1}^{I} 2^{-E_{A,i}}$$

and $E_A = -\log_2(P_A)$, assuming $I \ll 2^{E_A}$. Bob computes E_B similarly. Alice and Bob exchange their E_z values and each computes $E = \min(E_A, E_B)$. That value, E, is the numeric result of the successful round. The individual entropy computation must be conditional on all demonstrated knowledge so far in the protocol. That is, if Mallet has succeeded in guessing prior question/answer pairs, then the fact that the protocol continues gives him confirmation of his guess(es).

The probability of a successful guess by Mallet is, to a first order approximation, 2^{-E}. Since these entropies are computed conditional on previous rounds, the sum, X, of E values from all rounds so far gives the the probability of an eavesdropper's guessing the whole protocol so far to be 2^{-X}.

Answer entropy magnitude

Common-knowledge answers which are short enough to type uniquely (or nearly so) include names of people as possibly the simplest class. One sample of names from the employee list of a medium size company was analyzed to get an estimate of the entropy of such common knowledge answers. That analysis produced the results in the table below:

Items	N	$\log_2(N)$	Entropy
People	2507	11.292	
Full names	2501	11.288	11.287
Last names	2012	10.974	10.796
First names	802	9.647	8.376

Interlock protocols

If Alice and Bob were merely to transmit T_x to each other, Mallet would receive each. Since we are dealing with low entropy individual answers, Mallet could perform a dictionary attack, given T_x, and determine R_x. Given R_x, he could form the correct response, using keys he had provided to the other party, and allow the protocol to succeed.

Rivest and Shamir designed and published an interlock protocol[5] to prevent such attacks. Instead of transmitting the entire message, Alice and Bob each transmit one half of the message[8], waiting to receive the first half before transmitting the second half. Unfortunately, this works only when large entropy secrets are exchanged. If the answers are of small entropy, as in our case, Mallet can do a dictionary attack based on half the message as easily as on the whole message.

The interlock protocol can be blinded. Torben Pedersen[6] proposed one such scheme. If the secret is a, one chooses a high-entropy random u and computes $x = g^a h^u \bmod p$ where p is prime, g and h are generators of the group mod p, and $\log_g(h)$ is unknown. The first exchange carries x while the second exchange carries (a, u).

As a variant on Pedersen's interlock, one can use $x = H(a|u)$ where $H()$ is a presumed one-way function (e.g., a strong cryptographic hash).

Bellovin and Merritt[1] have presented an attack on the Rivest and Shamir interlock protocol which permits access to one side of the conversation and, if shared secrets are re-used, eventually to both sides. That attack works as well for the Pedersen commitment mechanism.

If communication is cheap compared to computation, there is a variant of the interlock protocol which is less vulnerable to the Bellovin and Merritt attack. Alice and Bob can release T_x to each other one bit at a time. Under this interlock protocol, once an accumulating partial result no longer matches any

[8] assuming the message is a single unit, such as a single RSA encryption or a single hash function output block

of the legal possible results, the protocol round is terminated in failure.

Probability of guessing a round

Using the basic Rivest-Shamir interlock protocol, it is possible for Mallet to perform a dictionary search over all possible answers, compute the hash for each and match half the resulting hash to the one received. The probability of collision, that two different dictionary entries would result in the same hash value, is effectively 0 assuming $E \ll |H|$, so that Mallet's probability of success is nearly 1.

Using Torben Pedersen's interlock, a dictionary attack is thwarted through blinding by the unique random value, u. Mallet can still perform the Bellovin and Merritt attack. Assume in the example below that of the two challenges offered, Bob's is the easier for Mallet to guess and therefore has the lesser entropy.

1. Alice generates u_A at random to mask secret a_A and computes $x_A = g^{a_A} h^{u_A} mod p$
2. Bob generates u_B at random to mask secret a_B and computes $x_B = g^{a_B} h^{u_B} mod p$
3. A→: x_A
4. B→: x_B
5. →M: x_A
6. →M: x_B
7. Mallet generates u'_B at random to mask secret a'_B (derived from his guess for Bob's answer combined with the keys Mallet has established with Alice) and computes $x'_B = g^{a'_B} h^{u'_B} mod p$
8. M→A: x'_B
9. A→: (a_A, u_A)
10. →M: (a_A, u_A)
11. Mallet performs a dictionary attack on a_A in order to learn the underlying secret and uses that secret to form x'_A to send on to Bob.
12. Mallet and Bob finish authenticating one another
13. M→A: (a'_B, u'_B), which will be accepted by Alice with probability 2^{-E_B} where E_B is the entropy of Bob's answer.

If $E = \min(E_A, E_B)$, the probability of Mallet's success is 2^{-E} using bit-at-a-time interlock, just as it was under the Pedersen interlock. That is, Mallet chooses one of the two answers to guess and constructs his reply accordingly. If he guesses correctly, then he can learn from that participant the full answer for the other participant – through a dictionary attack on a. He will guess correctly with probability 2^{-E}. Once he has the correct answer from one participant, he can engage in the protocol with the other participant.

However, should Mallet fail to guess correctly, the round will have failed and he will have learned only a limited number of bits of a legitimate answer. This would ease his completion of the other side of the protocol, but that round has failed and he has no purpose for finishing the other side. The note in the previous section that this variant is less vulnerable to the Bellovin and Merrit attack refers to this reduction of the information leakage to Mallet.

Adjustments to entropy

It is envisioned that this protocol would be used only once, when old friends meet on-line and share their first signature key. Assuming they were successful in having a secure channel, they can rest assured that the small secrets they exchanged to verify one another's identity and the security of the channel remain their secrets.

However, if they discover that the protocol failed, they must assume that Mallet has intercepted some number of secrets (or bits about secrets, if the bit-interlock protocol was used), and record the compromise of those secrets, adjusting their entropy (to 0, for cases where Mallet correctly guessed or appeared to guess; reduced by the number of released bits, for cases where Mallet failed). Those adjusted entropies must be used for any subsequent execution of this protocol.

References

[1] Bellovin and Merritt, "An Attack on the *Interlock Protocol* When Used for Authentication", IEEE Trans. Inf. Theory 40:1, Jan 1994.

[2] Ellison, "Generalized Certificates", manuscript, http://www.clark.net/pub/cme/html/cert.html

[3] McConnell and Appel, "Enabling Privacy, Commerce, Security and Public Safety in the Global Information Infrastructure", report of

the Interagency Working Group on Cryptography Policy, May 12, 1996; [quote from paragraph 5 of the Introduction]

[4] Rivest and Lampson, "SDSI – A Simple Distributed Security Infrastructure", manuscript, http://theory.lcs.mit.edu/~rivest/sdsi.ps

[5] Rivest and Shamir, "How to Expose an Eavesdropper", CACM, Vol. 27, April 1984, pp. 393-395.

[6] Torben Pryds Pedersen, "Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing", Advances in Cryptology - CRYPTO '91, LNCS 576, pp. 129-140.

[7] *Webster's Ninth New Collegiate Dictionary*, Merriam-Webster Inc., Springfield MA, 1991.

Secure Deletion of Data from Magnetic and Solid-State Memory

Peter Gutmann

Department of Computer Science
University of Auckland
pgut001@cs.auckland.ac.nz

Abstract

With the use of increasingly sophisticated encryption systems, an attacker wishing to gain access to sensitive data is forced to look elsewhere for information. One avenue of attack is the recovery of supposedly erased data from magnetic media or random-access memory. This paper covers some of the methods available to recover erased data and presents schemes to make this recovery significantly more difficult.

1. Introduction

Much research has gone into the design of highly secure encryption systems intended to protect sensitive information. However work on methods of securing (or at least safely deleting) the original plaintext form of the encrypted data against sophisticated new analysis techniques seems difficult to find. In the 1980's some work was done on the recovery of erased data from magnetic media [1] [2] [3], but to date the main source of information is government standards covering the destruction of data. There are two main problems with these official guidelines for sanitizing media. The first is that they are often somewhat old and may predate newer techniques for both recording data on the media and for recovering the recorded data. For example most of the current guidelines on sanitizing magnetic media predate the early-90's jump in recording densities, the adoption of sophisticated channel coding techniques such as PRML, the use of magnetic force microscopy for the analysis of magnetic media, and recent studies of certain properties of magnetic media recording such as the behaviour of erase bands. The second problem with official data destruction standards is that the information in them may be partially inaccurate in an attempt to fool opposing intelligence agencies (which is probably why a great many guidelines on sanitizing media are classified). By deliberately under-stating the requirements for media sanitization in publicly-available guides, intelligence agencies can preserve their information-gathering capabilities while at the same time protecting their own data using classified techniques.

This paper represents an attempt to analyse the problems inherent in trying to erase data from magnetic disk media and random-access memory without access to specialised equipment, and suggests methods for ensuring that the recovery of data from these media can be made as difficult as possible for an attacker.

2. Methods of Recovery for Data stored on Magnetic Media

Magnetic force microscopy (MFM) is a recent technique for imaging magnetization patterns with high resolution and minimal sample preparation. The technique is derived from scanning probe microscopy (SPM) and uses a sharp magnetic tip attached to a flexible cantilever placed close to the surface to be analysed, where it interacts with the stray field emanating from the sample. An image of the field at the surface is formed by moving the tip across the surface and measuring the force (or force gradient) as a function of position. The strength of the interaction is measured by monitoring the position of the cantilever using an optical interferometer or tunnelling sensor.

Magnetic force scanning tunneling microscopy (STM) is a more recent variant of this technique which uses a probe tip typically made by plating pure nickel onto a prepatterned surface, peeling the resulting thin film from the substrate it was plated onto and plating it with a thin layer of gold to minimise corrosion, and mounting it in a probe where it is placed at some small bias potential (typically a few tenths of a nanoamp at a few volts DC) so that electrons from the surface under test can tunnel across the gap to the probe tip (or vice versa). The probe is scanned across the surface to be

analysed as a feedback system continuously adjusts the vertical position to maintain a constant current. The image is then generated in the same way as for MFM [4] [5].

Other techniques which have been used in the past to analyse magnetic media are the use of ferrofluid in combination with optical microscopes (which, with gigabit/square inch recording density is no longer feasible as the magnetic features are smaller than the wavelength of visible light) and a number of exotic techniques which require significant sample preparation and expensive equipment. In comparison, MFM can be performed through the protective overcoat applied to magnetic media, requires little or no sample preparation, and can produce results in a very short time.

Even for a relatively inexperienced user the time to start getting images of the data on a drive platter is about 5 minutes. To start getting useful images of a particular track requires more than a passing knowledge of disk formats, but these are well-documented, and once the correct location on the platter is found a single image would take approximately 2-10 minutes depending on the skill of the operator and the resolution required. With one of the more expensive MFM's it is possible to automate a collection sequence and theoretically possible to collect an image of the entire disk by changing the MFM controller software.

There are, from manufacturers sales figures, several thousand SPM's in use in the field today, some of which have special features for analysing disk drive platters, such as the vacuum chucks for standard disk drive platters along with specialised modes of operation for magnetic media analysis. These SPM's can be used with sophisticated programmable controllers and analysis software to allow automation of the data recovery process. If commercially-available SPM's are considered too expensive, it is possible to build a reasonably capable SPM for about US$1400, using a PC as a controller [6].

Faced with techniques such as MFM, truly deleting data from magnetic media is very difficult. The problem lies in the fact that when data is written to the medium, the write head sets the polarity of most, but not all, of the magnetic domains. This is partially due to the inability of the writing device to write in exactly the same location each time, and partially due to the variations in media sensitivity and field strength over time and among devices.

In conventional terms, when a one is written to disk the media records a one, and when a zero is written the media records a zero. However the actual effect is closer to obtaining a 0.95 when a zero is overwritten with a one, and a 1.05 when a one is overwritten with a one. Normal disk circuitry is set up so that both these values are read as ones, but using specialised circuitry it is possible to work out what previous "layers" contained. The recovery of at least one or two layers of overwritten data isn't too hard to perform by reading the signal from the analog head electronics with a high-quality digital sampling oscilloscope, downloading the sampled waveform to a PC, and analysing it in software to recover the previously recorded signal. What the software does is generate an "ideal" read signal and subtract it from what was actually read, leaving as the difference the remnant of the previous signal. Since the analog circuitry in a commercial hard drive is nowhere near the quality of the circuitry in the oscilloscope used to sample the signal, the ability exists to recover a lot of extra information which isn't exploited by the hard drive electronics (although with newer channel coding techniques such as PRML (explained further on) which require extensive amounts of signal processing, the use of simple tools such as an oscilloscope to directly recover the data is no longer possible).

Using MFM, we can go even further than this. During normal readback, a conventional head averages the signal over the track, and any remnant magnetization at the track edges simply contributes a small percentage of noise to the total signal. The sampling region is too broad to distinctly detect the remnant magnetization at the track edges, so that the overwritten data which is still present beside the new data cannot be recovered without the use of specialised techniques such as MFM or STM (in fact one of the "official" uses of MFM or STM is to evaluate the effectiveness of disk drive servo-positioning mechanisms) [7]. Most drives are capable of microstepping the heads for internal diagnostic and error recovery purposes (typical error recovery strategies consist of rereading tracks with slightly changed data threshold and window offsets and varying the head positioning by a few percent to either side of the track), but writing to the media while the head is off-track in order to erase the remnant signal carries too much risk of making neighbouring tracks unreadable to be useful (for this reason the microstepping capability is made very difficult to access by external means).

These specialised techniques also allow data to be recovered from magnetic media long after the read/write head of the drive is incapable of reading

anything useful. For example one experiment in AC erasure involved driving the write head with a 40 MHz square wave with an initial current of 12 mA which was dropped in 2 mA steps to a final level of 2 mA in successive passes, an order of magnitude more than the usual write current which ranges from high microamps to low milliamps. Any remnant bit patterns left by this erasing process were far too faint to be detected by the read head, but could still be observed using MFM [8].

Even with a DC erasure process, traces of the previously recorded signal may persist until the applied DC field is several times the media coercivity [9].

Deviations in the position of the drive head from the original track may leave significant portions of the previous data along the track edge relatively untouched. Newly written data, present as wide alternating light and dark bands in MFM and STM images, are often superimposed over previously recorded data which persists at the track edges. Regions where the old and new data coincide create continuous magnetization between the two. However, if the new transition is out of phase with the previous one, a few microns of *erase band* with no definite magnetization are created at the juncture of the old and new tracks. The write field in the erase band is above the coercivity of the media and would change the magnetization in these areas, but its magnitude is not high enough to create new well-defined transitions. One experiment involved writing a fixed pattern of all 1's with a bit interval of 2.5 um, moving the write head off-track by approximately half a track width, and then writing the pattern again with a frequency slightly higher than that of the previously recorded track for a bit interval of 2.45 μm to create all possible phase differences between the transitions in the old and new tracks. Using a 4.2 μm wide head produced an erase band of approximately 1 μm in width when the old and new tracks were 180° out of phase, dropping to almost nothing when the two tracks were in-phase. Writing data at a higher frequency with the original tracks bit interval at 0.5 μm and the new tracks bit interval at 0.49 μm allows a single MFM image to contain all possible phase differences, showing a dramatic increase in the width of the erase band as the two tracks move from in-phase to 180° out of phase [10].

In addition, the new track width can exhibit modulation which depends on the phase relationship between the old and new patterns, allowing the previous data to be recovered even if the old data patterns themselves are no longer distinct. The overwrite performance also depends on the position of the write head relative to the originally written track. If the head is directly aligned with the track, overwrite performance is relatively good; as the head moves offtrack, the performance drops markedly as the remnant components of the original data are read back along with the newly-written signal. This effect is less noticeable as the write frequency increases due to the greater attenuation of the field with distance [11].

When all the above factors are combined it turns out that each track contains an image of everything ever written to it, but that the contribution from each "layer" gets progressively smaller the further back it was made. Intelligence organisations have a *lot* of expertise in recovering these palimpsestuous images.

3. Erasure of Data stored on Magnetic Media

The general concept behind an overwriting scheme is to flip each magnetic domain on the disk back and forth as much as possible (this is the basic idea behind degaussing) without writing the same pattern twice in a row. If the data was encoded directly, we could simply choose the desired overwrite pattern of ones and zeroes and write it repeatedly. However, disks generally use some form of run-length limited (RLL) encoding, so that the adjacent ones won't be written. This encoding is used to ensure that transitions aren't placed too closely together, or too far apart, which would mean the drive would lose track of where it was in the data.

To erase magnetic media, we need to overwrite it many times with alternating patterns in order to expose it to a magnetic field oscillating fast enough that it does the desired flipping of the magnetic domains in a reasonable amount of time. Unfortunately, there is a complication in that we need to saturate the disk surface to the greatest depth possible, and very high frequency signals only "scratch the surface" of the magnetic medium. Disk drive manufacturers, in trying to achieve ever-higher densities, use the highest possible frequencies, whereas we really require the lowest frequency a disk drive can produce. Even this is still rather high. The best we can do is to use the lowest frequency possible for overwrites, to penetrate as deeply as possible into the recording medium.

The write frequency also determines how effectively previous data can be overwritten due to the dependence of the field needed to cause magnetic switching on the length of time the field is applied. Tests on a number of typical disk drive heads have shown a difference of

up to 20 dB in overwrite performance when data recorded at 40 kFCI (flux changes per inch), typical of recent disk drives, is overwritten with a signal varying from 0 to 100 kFCI. The best average performance for the various heads appears to be with an overwrite signal of around 10 kFCI, with the worst performance being at 100 kFCI [12]. The track write width is also affected by the write frequency — as the frequency increases, the write width decreases for both MR and TFI heads. In [13] there was a decrease in write width of around 20% as the write frequency was increased from 1 to 40 kFCI, with the decrease being most marked at the high end of the frequency range. However, the decrease in write width is balanced by a corresponding increase in the two side-erase bands so that the sum of the two remains nearly constant with frequency and equal to the DC erase width for the head. The media coercivity also affects the width of the write and erase bands, with their width dropping as the coercivity increases (this is one of the explanations for the ever-increasing coercivity of newer, higher-density drives).

To try to write the lowest possible frequency we must determine what decoded data to write to produce a low-frequency encoded signal.

In order to understand the theory behind the choice of data patterns to write, it is necessary to take a brief look at the recording methods used in disk drives. The main limit on recording density is that as the bit density is increased, the peaks in the analog signal recorded on the media are read at a rate which may cause them to appear to overlap, creating intersymbol interference which leads to data errors. Traditional peak detector read channels try to reduce the possibility of intersymbol interference by coding data in such a way that the analog signal peaks are separated as far as possible. The read circuitry can then accurately detect the peaks (actually the head itself only detects *transitions* in magnetisation, so the simplest recording code uses a transition to encode a 1 and the absence of a transition to encode a 0. The transition causes a positive/negative peak in the head output voltage (thus the name "peak detector read channel"). To recover the data, we differentiate the output and look for the zero crossings). Since a long string of 0's will make clocking difficult, we need to set a limit on the maximum consecutive number of 0's. The separation of peaks is implemented as some form of run-length-limited, or RLL, coding.

The RLL encoding used in most current drives is described by pairs of run-length limits (d, k), where d is the minimum number of 0 symbols which must occur between each 1 symbol in the encoded data, and k is the maximum. The parameters (d, k) are chosen to place adjacent 1's far enough apart to avoid problems with intersymbol interference, but not so far apart that we lose synchronisation.

The grandfather of all RLL codes was FM, which wrote one user data bit followed by one clock bit, so that a 1 bit was encoded as two transitions (1 wavelength) while a 0 bit was encoded as one transition (½ wavelength). A different approach was taken in modified FM (MFM), which suppresses the clock bit except between adjacent 0's (the ambiguity in the use of the term MFM is unfortunate. From here on it will be used to refer to modified FM rather than magnetic force microscopy). Taking three example sequences `0000`, `1111`, and `1010`, these will be encoded as `0(1)0(1)0(1)0`, `1(0)1(0)1(0)1`, and `1(0)0(0)1(0)0` (where the ()s are the clock bits inserted by the encoding process). The maximum time between 1 bits is now three 0 bits (so that the peaks are no more than four encoded time periods apart), and there is always at least one 0 bit (so that the peaks in the analog signal are at least two encoded time periods apart), resulting in a (1,3) RLL code. (1,3) RLL/MFM is the oldest code still in general use today, but is only really used in floppy drives which need to remain backwards-compatible.

These constraints help avoid intersymbol interference, but the need to separate the peaks reduces the recording density and therefore the amount of data which can be stored on a disk. To increase the recording density, MFM was gradually replaced by (2,7) RLL (the original "RLL" format), and that in turn by (1,7) RLL, each of which placed less constraints on the recorded signal.

Using our knowledge of how the data is encoded, we can now choose which decoded data patterns to write in order to obtain the desired encoded signal. The three encoding methods described above cover the vast majority of magnetic disk drives. However, each of these has several possible variants. With MFM, only one is used with any frequency, but the newest (1,7) RLL code has at least half a dozen variants in use. For MFM with at most four bit times between transitions, the lowest write frequency possible is attained by writing the repeating decoded data patterns `1010` and `0101`. These have a 1 bit every other "data" bit, and the intervening "clock" bits are all 0. We would also like patterns with every other clock bit set to 1 and all others set to 0, but these are not possible in the MFM encoding (such "violations" are used to generate special

marks on the disk to identify sector boundaries). The best we can do here is three bit times between transitions, which is generated by repeating the decoded patterns `100100`, `010010` and `001001`. We should use several passes with these patterns, as MFM drives are the oldest, lowest-density drives around (this is especially true for the very-low-density floppy drives). As such, they are the easiest to recover data from with modern equipment and we need to take the most care with them.

From MFM we jump to the next simplest case, which is (1,7) RLL. Although there can be as many as 8 bit times between transitions, the lowest sustained frequency we can have in practice is 6 bit times between transitions. This is a desirable property from the point of view of the clock-recovery circuitry, and all (1,7) RLL codes seem to have this property. We now need to find a way to write the desired pattern without knowing the particular (1,7) RLL code used. We can do this by looking at the way the drives error-correction system works. The error-correction is applied to the decoded data, even though errors generally occur in the encoded data. In order to make this work well, the data encoding should have limited error amplification, so that an erroneous encoded bit should affect only a small, finite number of decoded bits.

Decoded bits therefore depend only on nearby encoded bits, so that a repeating pattern of encoded bits will correspond to a repeating pattern of decoded bits. The repeating pattern of encoded bits is 6 bits long. Since the rate of the code is $^2/_3$, this corresponds to a repeating pattern of 4 decoded bits. There are only 16 possibilities for this pattern, making it feasible to write all of them during the erase process. So to achieve good overwriting of (1,7) RLL disks, we write the patterns `0000`, `0001`, `0010`, `0011`, `0100`, `0101`, `0110`, `0111`, `1000`, `1001`, `1010`, `1011`, `1100`, `1101`, `1110`, and `1111`. These patterns also conveniently cover two of the ones needed for MFM overwrites, although we should add a few more iterations of the MFM-specific patterns for the reasons given above.

Finally, we have (2,7) RLL drives. These are similar to MFM in that an eight-bit-time signal can be written in some phases, but not all. A six-bit-time signal will fill in the remaining cracks. Using a ½ encoding rate, an eight-bit-time signal corresponds to a repeating pattern of 4 data bits. The most common (2,7) RLL code is shown in Table 1.

Decoded Data	(2,7) RLL Encoded Data
00	1000
01	0100
100	001000
101	100100
111	000100
1100	00001000
1101	00100100

Table 1: The most common (2,7) RLL Code

The second most common (2,7) RLL code is the same but with the "decoded data" complemented, which doesn't alter these patterns. Writing the required encoded data can be achieved for every other phase using patterns of `0x33`, `0x66`, `0xCC` and `0x99`, which are already written for (1,7) RLL drives.

Six-bit-time patterns can be written using 3-bit repeating patterns. The all-zero and all-one patterns overlap with the (1,7) RLL patterns, leaving six others:

```
001001001001001001001001
   2   4   9   2   4   9
```

in binary or `0x24 0x92 0x49`, `0x92 0x49 0x24` and `0x49 0x24 0x92` in hex, and

```
011011011011011011011011
   6   D   B   6   D   B
```

in binary or `0x6D 0xB6 0xDB`, `0xB6 0xDB 0x6D` and `0xDB 0x6D 0xB6` in hex. The first three are the same as the MFM patterns, so we need only three extra patterns to cover (2,7) RLL drives.

Although (1,7) is more popular in recent (post-1990) drives, some older hard drives do still use (2,7) RLL, and with the ever-increasing reliability of newer drives it is likely that they will remain in use for some time to come, often being passed down from one machine to another. The above three patterns also cover any problems with endianness issues, which weren't a concern in the previous two cases, but would be in this case (actually, thanks to the strong influence of IBM mainframe drives, everything seems to be uniformly big-endian within bytes, with the most significant bit being written to the disk first).

The latest high-density drives use methods like Partial-Response Maximum-Likelihood (PRML) encoding, which may be roughly equated to the trellis encoding done by V.32 modems in that it is effective but

computationally expensive. PRML codes are still RLL codes, but with somewhat different constraints. A typical code might have (0,4,4) constraints in which the 0 means that 1's in a data stream can occur right next to 0's (so that peaks in the analog readback signal are not separated), the first 4 means that there can be no more than four 0's between 1's in a data stream, and the second 4 specifies the maximum number of 0's between 1's in certain symbol subsequences. PRML codes avoid intersymbol influence errors by using digital filtering techniques to shape the read signal to exhibit desired frequency and timing characteristics (this is the "partial response" part of PRML) followed by maximum-likelihood digital data detection to determine the most likely sequence of data bits that was written to the disk (this is the "maximum likelihood" part of PRML).

PRML channels achieve the same low bit error rate as standard peak-detection methods, but with much higher recording densities, while using the same heads and media. Several manufacturers are currently engaged in moving their peak-detection-based product lines across to PRML, giving a 30-40% density increase over standard RLL channels [14].

Since PRML codes don't try to separate peaks in the same way that non-PRML RLL codes do, all we can do is to write a variety of random patterns because the processing inside the drive is too complex to second-guess. Fortunately, these drives push the limits of the magnetic media much more than older drives ever did by encoding data with much smaller magnetic domains, closer to the physical capacity of the magnetic media (the current state of the art in PRML drives has a track density of around 6700 TPI (tracks per inch) and a data recording density of 170 kFCI, nearly double that of the nearest (1,7) RLL equivalent. A convenient side-effect of these very high recording densities is that a written transition may experience the write field cycles for successive transitions, especially at the track edges where the field distribution is much broader [15]. Since this is also where remnant data is most likely to be found, this can only help in reducing the recoverability of the data). If these drives require sophisticated signal processing just to read the most recently written data, reading overwritten layers is also correspondingly more difficult. A good scrubbing with random data will do about as well as can be expected.

We now have a set of 22 overwrite patterns which should erase everything, regardless of the raw encoding. The basic disk eraser can be improved slightly by adding random passes before and after the erase process, and by performing the deterministic passes in random order to make it more difficult to guess which of the known data passes were made at which point. To deal with all this in the overwrite process, we use the sequence of 35 consecutive writes shown in Table 2.

The MFM-specific patterns are repeated twice because MFM drives have the lowest density and are thus particularly easy to examine. The deterministic patterns between the random writes are permuted before the write is performed, to make it more difficult for an opponent to use knowledge of the erasure data written to attempt to recover overwritten data (in fact we need to use a cryptographically strong random number generator to perform the permutations to avoid the problem of an opponent who can read the last overwrite pass being able to predict the previous passes and "echo cancel" passes by subtracting the known overwrite data).

If the device being written to supports caching or buffering of data, this should be disabled to ensure that physical disk writes are performed for each pass instead of everything but the last pass being lost in the buffering. For example physical disk access can be forced during SCSI-2 Group 1 write commands by setting the Force Unit Access bit in the SCSI command block (although at least one popular drive has a bug which causes all writes to be ignored when this bit is set — remember to test your overwrite scheme before you deploy it). Another consideration which needs to be taken into account when trying to erase data through software is that drives conforming to some of the higher-level protocols such as the various SCSI standards are relatively free to interpret commands sent to them in whichever way they choose (as long as they still conform to the SCSI specification). Thus some drives, if sent a FORMAT UNIT command may return immediately without performing any action, may simply perform a read test on the entire disk (the most common option), or may actually write data to the disk (the SCSI-2 standard includes an initialization pattern (IP) option for the FORMAT UNIT command, however this is not necessarily supported by existing drives).

If the data is very sensitive and is stored on floppy disk, it can best be destroyed by removing the media from the disk liner and burning it, or by burning the entire disk, liner and all (most floppy disks burn remarkably well — albeit with quantities of oily smoke — and leave very little residue).

Pass No.	Data Written	Encoding Scheme Targeted		
1	Random			
2	Random			
3	Random			
4	Random			
5	01010101 01010101 01010101 0x55	(1,7) RLL		MFM
6	10101010 10101010 10101010 0xAA	(1,7) RLL		MFM
7	10010010 01001001 00100100 0x92 0x49 0x24		(2,7) RLL	MFM
8	01001001 00100100 10010010 0x49 0x24 0x92		(2,7) RLL	MFM
9	00100100 10010010 01001001 0x24 0x92 0x49		(2,7) RLL	MFM
10	00000000 00000000 00000000 0x00	(1,7) RLL	(2,7) RLL	
11	00010001 00010001 00010001 0x11	(1,7) RLL		
12	00100010 00100010 00100010 0x22	(1,7) RLL		
13	00110011 00110011 00110011 0x33	(1,7) RLL	(2,7) RLL	
14	01000100 01000100 01000100 0x44	(1,7) RLL		
15	01010101 01010101 01010101 0x55	(1,7) RLL		MFM
16	01100110 01100110 01100110 0x66	(1,7) RLL	(2,7) RLL	
17	01110111 01110111 01110111 0x77	(1,7) RLL		
18	10001000 10001000 10001000 0x88	(1,7) RLL		
19	10011001 10011001 10011001 0x99	(1,7) RLL	(2,7) RLL	
20	10101010 10101010 10101010 0xAA	(1,7) RLL		MFM
21	10111011 10111011 10111011 0xBB	(1,7) RLL		
22	11001100 11001100 11001100 0xCC	(1,7) RLL	(2,7) RLL	
23	11011101 11011101 11011101 0xDD	(1,7) RLL		
24	11101110 11101110 11101110 0xEE	(1,7) RLL		
25	11111111 11111111 11111111 0xFF	(1,7) RLL	(2,7) RLL	
26	10010010 01001001 00100100 0x92 0x49 0x24		(2,7) RLL	MFM
27	01001001 00100100 10010010 0x49 0x24 0x92		(2,7) RLL	MFM
28	00100100 10010010 01001001 0x24 0x92 0x49		(2,7) RLL	MFM
29	01101101 10110110 11011011 0x6D 0xB6 0xDB		(2,7) RLL	
30	10110110 11011011 01101101 0xB6 0xDB 0x6D		(2,7) RLL	
31	11011011 01101101 10110110 0xDB 0x6D 0xB6		(2,7) RLL	
32	Random			
33	Random			
34	Random			
35	Random			

Table 2: Overwrite Data

4. Other Methods of Erasing Magnetic Media

The previous section has concentrated on erasure methods which require no specialised equipment to perform the erasure. Alternative means of erasing media which do require specialised equipment are degaussing (a process in which the recording media is returned to its initial state) and physical destruction. Degaussing is a reasonably effective means of purging data from magnetic disk media, and will even work through most drive cases (research has shown that the aluminium housings of most disk drives attenuate the degaussing field by only about 2 dB [16]).

The switching of a single-domain magnetic particle from one magnetization direction to another requires the overcoming of an energy barrier, with an external magnetic field helping to lower this barrier. The switching depends not only on the magnitude of the external field, but also on the length of time for which it is applied. For typical disk drive media, the short-term field needed to flip enough of the magnetic domains to be useful in recording a signal is about 1/3 higher than the coercivity of the media (the exact figure varies with different media types) [17].

However, to effectively erase a medium to the extent that recovery of data from it becomes uneconomical requires a magnetic force of about five times the coercivity of the medium [18], although even small external magnetic fields are sufficient to upset the normal operation of a hard disk (typically a few gauss at DC, dropping to a few milligauss at 1 MHz). Coercivity (measured in Oersteds, Oe) is a property of magnetic material and is defined as the amount of magnetic field necessary to reduce the magnetic induction in the material to zero — the higher the coercivity, the harder it is to erase data from a medium. Typical figures for various types of magnetic media are given in Table 3 below.

Medium	Coercivity
5.25" 360K floppy disk	300 Oe
5.25" 1.2M floppy disk	675 Oe
3.5" 720K floppy disk	300 Oe
3.5" 1.44M floppy disk	700 Oe
3.5" 2.88M floppy disk	750 Oe
3.5" 21M floptical disk	750 Oe
Older (1980's) hard disks	900-1400 Oe
Newer (1990's) hard disks	1400-2200 Oe
1/2" magnetic tape	300 Oe
1/4" QIC tape	550 Oe
8 mm metallic particle tape	1500 Oe
DAT metallic particle tape	1500 Oe

Table 3: Typical Media Coercivity Figures

US Government guidelines class tapes of 350 Oe coercivity or less as low-energy or Class I tapes and tapes of 350-750 Oe coercivity as high-energy or Class II tapes. Degaussers are available for both types of tapes. Tapes of over 750 Oe coercivity are referred to as Class III, with no known degaussers capable of fully erasing them being known [19], since even the most powerful commercial AC degausser cannot generate the recommended 7,500 Oe needed for full erasure of a typical DAT tape currently used for data backups.

Degaussing of disk media is somewhat more difficult — even older hard disks generally have a coercivity equivalent to Class III tapes, making them fairly difficult to erase at the outset. Since manufacturers rate their degaussers in peak gauss and measure the field at a certain orientation which may not be correct for the type of medium being erased, and since degaussers tend to be rated by whether they erase sufficiently for clean rerecording rather than whether they make the information impossible to recover, it may be necessary to resort to physical destruction of the media to completely sanitise it (in fact since degaussing destroys the sync bytes, ID fields, error correction information, and other paraphernalia needed to identify sectors on the media, thus rendering the drive unusable, it makes the degaussing process mostly equivalent to physical destruction). In addition, like physical destruction, it requires highly specialised equipment which is expensive and difficult to obtain (one example of an adequate degausser was the 2.5 MW Navy research magnet used by a former Pentagon site manager to degauss a 14" hard drive for 1½ minutes. It bent the platters on the drive and probably succeeded in erasing it beyond the capabilities of any data recovery attempts [20]).

5. Further Problems with Magnetic Media

A major issue which cannot be easily addressed using any standard software-based overwrite technique is the problem of defective sector handling. When the drive is manufactured, the surface is scanned for defects which are added to a defect list or flaw map. If further defects, called grown defects, occur during the life of the drive, they are added to the defect list by the drive or by drive management software. There are several techniques which are used to mask the defects in the defect list. The first, alternate tracks, moves data from tracks with defects to known good tracks. This scheme is the simplest, but carries a high access cost, as each read from a track with defects requires seeking to the alternate track and a rotational latency delay while waiting for the data location to appear under the head, performing the read or write, and, if the transfer is to continue onto a neighbouring track, seeking back to the original position. Alternate tracks may be interspersed among data tracks to minimise the seek time to access them.

A second technique, alternate sectors, allocates alternate sectors at the end of the track to minimise seeks caused by defective sectors. This eliminates the seek delay, but still carries some overhead due to rotational latency. In addition it reduces the usable storage capacity by 1-3%.

A third technique, inline sector sparing, again allocates a spare sector at the end of each track, but resequences the sector ID's to skip the defective sector and include the spare sector at the end of the track, in effect pushing the sectors past the defective one towards the end of the track. The associated cost is the lowest of the three, being one sector time to skip the defective sector [21].

The handling of mapped-out sectors and tracks is an issue which can't be easily resolved without the cooperation of hard drive manufacturers. Although some SCSI and IDE hard drives may allow access to defect lists and even to mapped-out areas, this must be done in a highly manufacturer- and drive-specific manner. For example the SCSI-2 READ DEFECT DATA command can be used to obtain a list of all defective areas on the drive. Since SCSI logical block numbers may be mapped to arbitrary locations on the disk, the defect list is recorded in terms of heads, tracks, and sectors. As all SCSI device addressing is performed in terms of logical block numbers, mapped-out sectors or tracks cannot be addressed. The only reasonably portable possibility is to clear various automatic correction flags in the read-write error recovery mode page to force the SCSI device to report read/write errors to the user instead of transparently remapping the defective areas. The user can then use the READ LONG and WRITE LONG commands (which allow access to sectors and extra data even in the presence of read/write errors), to perform any necessary operations on the defective areas, and then use the REASSIGN BLOCKS command to reassign the defective sections. However this operation requires an in-depth knowledge of the operation of the SCSI device and extensive changes to disk drivers, and more or less defeats the purpose of having an intelligent peripheral.

The ANSI X3T-10 and X3T-13 subcommittees are currently looking at creating new standards for a Universal Security Reformat command for IDE and SCSI hard disks which will address these issues. This will involve a multiple-pass overwrite process which covers mapped-out disk areas with deliberate off-track writing. Many drives available today can be modified for secure erasure through a firmware upgrade, and once the new firmware is in place the erase procedure is handled by the drive itself, making unnecessary any interaction with the host system beyond the sending of the command which begins the erase process.

Long-term ageing can also have a marked effect on the erasability of magnetic media. For example, some types of magnetic tape become increasingly difficult to erase after being stored at an elevated temperature or having contained the same magnetization pattern for a considerable period of time [22]. The same applies for magnetic disk media, with decreases in erasability of several dB being recorded [23]. The erasability of the data depends on the amount of time it has been stored on the media, not on the age of the media itself (so that, for example, a five-year-old freshly-written disk is no less erasable than a new freshly-written disk).

The dependence of media coercivity on temperature can affect overwrite capability if the data was initially recorded at a temperature where the coercivity was low (so that the recorded pattern penetrated deep into the media), but must be overwritten at a temperature where the coercivity is relatively high. This is important in hard disk drives, where the temperature varies depending on how long the unit has been used and, in the case of drives with power-saving features enabled, how recently and frequently it has been used. However the overwrite performance depends not only on temperature-dependent changes in the media, but also on temperature-dependent changes in the read/write head. Thankfully the combination of the most common media used in current drives with various common

types of read/write heads produce a change in overwrite performance of only a few hundredths of a decibel per degree over the temperature range -40°C to + 40°C, as changes in the head compensate for changes in the media [24].

Another issue which needs to be taken into account is the ability of most newer storage devices to recover from having a remarkable amount of damage inflicted on them through the use of various error-correction schemes. As increasing storage densities began to lead to multiple-bit errors, manufacturers started using sophisticated error-correction codes (ECC's) capable of correcting multiple error bursts. A typical drive might have 512 bytes of data, 4 bytes of CRC, and 11 bytes of ECC per sector. This ECC would be capable of correcting single burst errors of up to 22 bits or double burst errors of up to 11 bits, and can detect a single burst error of up to 51 bits or three burst errors of up to 11 bits in length [25]. Another drive manufacturer quotes the ability to correct up to 120 bits, or up to 32 bits on the fly, using 198-bit Reed-Solomon ECC [26]. Therefore even if some data is reliably erased, it may be possible to recover it using the built-in error-correction capabilities of the drive. Conversely, any erasure scheme which manages to destroy the ECC information (for example through the use of the SCSI-2 WRITE LONG command which can be used to write to areas of a disk sector outside the normal data areas) stands a greater chance of making the data unrecoverable.

6. Sidestepping the Problem

The easiest way to solve the problem of erasing sensitive information from magnetic media is to ensure that it never gets to the media in the first place. Although not practical for general data, it is often worthwhile to take steps to keep particularly important information such as encryption keys from ever being written to disk. This would typically happen when the memory containing the keys is paged out to disk by the operating system, where they can then be recovered at a later date, either manually or using software which is aware of the in-memory data format and can locate it automatically in the swap file (for example there exists software which will search the Windows swap file for keys from certain DOS encryption programs). An even worse situation occurs when the data is paged over a network, allowing anyone with a packet sniffer or similar tool on the same subnet to observe the information (for example there exists software which will monitor and even alter NFS traffic on the fly which could be modified to look for known in-memory data patterns moving to and from a networked swap disk [27]).

To solve these problems the memory pages containing the information can be locked to prevent them from being paged to disk or transmitted over a network. This approach is taken by at least one encryption library, which allocates all keying information inside protected memory blocks visible to the user only as opaque handles, and then optionally locks the memory (provided the underlying OS allows it) to prevent it from being paged [28]. The exact details of locking pages in memory depend on the operating system being used. Many Unix systems now support the `mlock()`/`munlock()` calls or have some alternative mechanism hidden among the `mmap()`-related functions which can be used to lock pages in memory. Unfortunately these operations require superuser priviledges because of their potential impact on system performance if large ranges of memory are locked. Other systems such as Microsoft Windows NT allow user processes to lock memory with the `VirtualLock()`/`VirtualUnlock()` calls, but limit the total number of regions which can be locked.

Most paging algorithms are relatively insensitive to having sections of memory locked, and can even relocate the locked pages (since the logical to physical mapping is invisible to the user), or can move the pages to a "safe" location when the memory is first locked. The main effect of locking pages in memory is to increase the minimum working set size which, taken in moderation, has little noticeable effect on performance. The overall effects depend on the operating system and/or hardware implementations of virtual memory. Most Unix systems have a global page replacement policy in which a page fault may be satisfied by any page frame. A smaller number of operating systems use a local page replacement policy in which pages are allocated from a fixed (or occasionally dynamically variable) number of page frames allocated on a per-process basis. This makes them much more sensitive to the effects of locking pages, since every locked page decreases the (finite) number of pages available to the process. On the other hand it makes the system as a whole less sensitive to the effects of one process locking a large number of pages. The main effective difference between the two is that under a local replacement policy a process can only lock a small fixed number of pages without affecting other processes, whereas under a global replacement policy the number of pages a process can lock is determined

on a system-wide basis and may be affected by other processes.

In practice neither of these allocation strategies seem to cause any real problems. Although any practical measurements are very difficult to perform since they vary wildly depending on the amount of physical memory present, paging strategy, operating system, and system load, in practice locking a dozen 1K regions of memory (which might be typical of a system on which a number of users are running programs such as mail encryption software) produced no noticeable performance degradation observable by system-monitoring tools. On machines such as network servers handling large numbers of secure connections (for example an HTTP server using SSL), the effects of locking large numbers of pages may be more noticeable.

7. Methods of Recovery for Data stored in Random-Access Memory

Contrary to conventional wisdom, "volatile" semiconductor memory does not entirely lose its contents when power is removed. Both static (SRAM) and dynamic (DRAM) memory retains some information on the data stored in it while power was still applied. SRAM is particularly susceptible to this problem, as storing the same data in it over a long period of time has the effect of altering the preferred power-up state to the state which was stored when power was removed. Older SRAM chips could often "remember" the previously held state for several days. In fact, it is possible to manufacture SRAM's which always have a certain state on power-up, but which can be overwritten later on — a kind of "writeable ROM".

DRAM can also "remember" the last stored state, but in a slightly different way. It isn't so much that the charge (in the sense of a voltage appearing across a capacitance) is retained by the RAM cells, but that the thin oxide which forms the storage capacitor dielectric is highly stressed by the applied field, or is not stressed by the field, so that the properties of the oxide change slightly depending on the state of the data. One thing that can cause a threshold shift in the RAM cells is ionic contamination of the cell(s) of interest, although such contamination is rarer now than it used to be because of robotic handling of the materials and because the purity of the chemicals used is greatly improved. However, even a perfect oxide is subject to having its properties changed by an applied field. When it comes to contaminants, sodium is the most common offender — it is found virtually everywhere, and is a fairly small (and therefore mobile) atom with a positive charge. In the presence of an electric field, it migrates towards the negative pole with a velocity which depends on temperature, the concentration of the sodium, the oxide quality, and the other impurities in the oxide such as dopants from the processing. If the electric field is zero and given enough time, this stress tends to dissipate eventually.

The stress on the cell is a cumulative effect, much like charging an RC circuit. If the data is applied for only a few milliseconds then there is very little "learning" of the cell, but if it is applied for hours then the cell will acquire a strong (relatively speaking) change in its threshold. The effects of the stress on the RAM cells can be measured using the built-in self test capabilities of the cells, which provide the ability to impress a weak voltage on a storage cell in order to measure its margin. Cells will show different margins depending on how much oxide stress has been present. Many DRAM's have undocumented test modes which allow some normal I/O pin to become the power supply for the RAM core when the special mode is active. These test modes are typically activated by running the RAM in a nonstandard configuration, so that a certain set of states which would not occur in a normally-functioning system has to be traversed to activate the mode. Manufacturers won't admit to such capabilities in their products because they don't want their customers using them and potentially rejecting devices which comply with their spec sheets, but have little margin beyond that.

A simple but somewhat destructive method to speed up the annihilation of stored bits in semiconductor memory is to heat it. Both DRAM's and SRAM's will lose their contents a lot more quickly at $T_{junction}$ = 140°C than they will at room temperature. Several hours at this temperature with no power applied will clear their contents sufficiently to make recovery difficult. Conversely, to extend the life of stored bits with the power removed, the temperature should be dropped below -60°C. Such cooling should lead to weeks, instead of hours or days, of data retention.

8. Erasure of Data stored in Random-Access Memory

Simply repeatedly overwriting the data held in DRAM with new data isn't nearly as effective as it is for magnetic media. The new data will begin stressing or relaxing the oxide as soon as it is written, and the oxide

will immediately begin to take a "set" which will either reinforce the previous "set" or will weaken it. The greater the amount of time that new data has existed in the cell, the more the old stress is "diluted", and the less reliable the information extraction will be. Generally, the rates of change due to stress and relaxation are in the same order of magnitude. Thus, a few microseconds of storing the opposite data to the currently stored value will have little effect on the oxide. Ideally, the oxide should be exposed to as much stress at the highest feasible temperature and for as long as possible to get the greatest "erasure" of the data. Unfortunately if carried too far this has a rather detrimental effect on the life expectancy of the RAM.

Therefore the goal to aim for when sanitising memory is to store the data for as long as possible rather than trying to change it as often as possible. Conversely, storing the data for as short a time as possible will reduce the chances of it being "remembered" by the cell. Based on tests on DRAM cells, a storage time of one second causes such a small change in threshold that it probably isn't detectable. On the other hand, one minute is probably detectable, and 10 minutes is certainly detectable.

The most practical solution to the problem of DRAM data retention is therefore to constantly flip the bits in memory to ensure that a memory cell never holds a charge long enough for it to be "remembered". While not practical for general use, it is possible to do this for small amounts of very sensitive data such as encryption keys. This is particularly advisable where keys are stored in the same memory location for long periods of time and control access to large amounts of information, such as keys used for transparent encryption of files on disk drives. The bit-flipping also has the convenient side-effect of keeping the page containing the encryption keys at the top of the queue maintained by the system's paging mechanism, greatly reducing the chances of it being paged to disk at some point.

9. Conclusion

Data overwritten once or twice may be recovered by subtracting what is expected to be read from a storage location from what is actually read. Data which is overwritten an arbitrarily large number of times can still be recovered provided that the new data isn't written to the same location as the original data (for magnetic media), or that the recovery attempt is carried out fairly soon after the new data was written (for RAM). For this reason it is effectively impossible to sanitise storage locations by simple overwriting them, no matter how many overwrite passes are made or what data patterns are written. However by using the relatively simple methods presented in this paper the task of an attacker can be made significantly more difficult, if not prohibitively expensive.

Acknowledgments

The author would like to thank Nigel Bree, Peter Fenwick, Andy Hospodor, Kevin Martinez, Colin Plumb, and Charles Preston for their advice and input during the preparation of this paper.

References

[1] "Emergency Destruction of Information Storing Media", M.Slusarczuk et al, Institute for Defense Analyses, December 1987.

[2] "A Guide to Understanding Data Remanence in Automated Information Systems", National Computer Security Centre, September 1991.

[3] "Detection of Digital Information from Erased Magnetic Disks", Venugopal Veeravalli, Masters thesis, Carnegie-Mellon University, 1987.

[4] "Magnetic force microscopy: General principles and application to longitudinal recording media", D.Rugar, H.Mamin, P.Guenther, S.Lambert, J.Stern, I.McFadyen, and T.Yogi, *Journal of Applied Physics*, **Vol.68**, **No.3** (August 1990), p.1169.

[5] "Tunneling-stabilized Magnetic Force Microscopy of Bit Tracks on a Hard Disk", Paul Rice and John Moreland, *IEEE Trans.on Magnetics*, **Vol.27**, **No.3** (May 1991), p.3452.

[6] "NanoTools: The Homebrew STM Page", Jim Rice, `http://www.skypoint.com/-members/jrice/STMWebPage.html`.

[7] "Magnetic Force Scanning Tunnelling Microscope Imaging of Overwritten Data", Romel Gomez, Amr Adly, Isaak Mayergoyz, Edward Burke, *IEEE Trans.on Magnetics*, **Vol.28**, **No.5** (September 1992), p.3141.

[8] "Comparison of Magnetic Fields of Thin-Film Heads and Their Corresponding Patterns Using Magnetic Force Microscopy", Paul Rice, Bill Hallett, and John Moreland, *IEEE Trans.on Magnetics*, **Vol.30**, **No.6** (November 1994), p.4248.

[9] "Computation of Magnetic Fields in Hysteretic Media", Amr Adly, Isaak Mayergoyz, Edward Burke, *IEEE Trans.on Magnetics*, **Vol.29**, **No.6** (November 1993), p.2380.

[10] "Magnetic Force Microscopy Study of Edge Overwrite Characteristics in Thin Film Media", Jian-Gang Zhu, Yansheng Luo, and Juren Ding, *IEEE Trans.on Magnetics*, **Vol.30**, **No.6** (November 1994), p.4242.

[11] "Microscopic Investigations of Overwritten Data", Romel Gomez, Edward Burke, Amr Adly, Isaak Mayergoyz, J.Gorczyca, *Journal of Applied Physics*, **Vol.73**, **No.10** (May 1993), p.6001.

[12] "Relationship between Overwrite and Transition Shift in Perpendicular Magnetic Recording", Hiroaki Muraoka, Satoshi Ohki, and Yoshihisa Nakamura, *IEEE Trans.on Magnetics*, **Vol.30**, **No.6** (November 1994), p.4272.

[13] "Effects of Current and Frequency on Write, Read, and Erase Widths for Thin-Film Inductive and Magnetoresistive Heads", Tsann Lin, Jodie Christner, Terry Mitchell, Jing-Sheng Gau, and Peter George, *IEEE Trans.on Magnetics*, **Vol.25**, **No.1** (January 1989), p.710.

[14] "PRML Read Channels: Bringing Higher Densities and Performance to New-Generation Hard Drives", Quantum Corporation, 1995.

[15] "Density and Phase Dependence of Edge Erase Band in MR/Thin Film Head Recording", Yansheng Luo, Terence Lam, Jian-Gang Zhu, *IEEE Trans.on Magnetics*, **Vol.31**, **No.6** (November 1995), p.3105.

[16] "A Guide to Understanding Data Remanence in Automated Information Systems", National Computer Security Centre, September 1991.

[17] "Time-dependant Magnetic Phenomena and Particle-size Effects in Recording Media", *IEEE Trans.on Magnetics*, **Vol.26**, **No.1** (January 1990), p.193.

[18] "The Data Dilemna", Charles Preston, *Security Management Journal*, February 1995.

[19] "Magnetic Tape Degausser", NSA/CSS Specification L14-4-A, 31 October 1985.

[20] "How many times erased does DoD want?", David Hayes, posting to `comp.periphs.scsi` newsgroup, 24 July 1991, message-ID `1991Jul24.050701.16005@sulaco.lonestar.org`.

[21] "The Changing Nature of Disk Controllers", Andrew Hospodor and Albert Hoagland, *Proceedings of the IEEE*, **Vol.81**, **No.4** (April 1993), p.586.

[22] "Annealing Study of the Erasability of High Energy Tapes", L.Lekawat, G.Spratt, and M.Kryder, *IEEE Trans.on Magnetics*, **Vol.29**, **No.6** (November 1993), p.3628.

[23] "The Effect of Aging on Erasure in Particulate Disk Media", K.Mountfield and M.Kryder, *IEEE Trans.on Magnetics*, **Vol.25**, **No 5** (September 1989), p.3638.

[24] "Overwrite Temperature Dependence for Magnetic Recording", Takayuki Takeda, Katsumichi Tagami, and Takaaki Watanabe, *Journal of Applied Physics*, **Vol.63**, **No.8** (April 1988), p.3438.

[25] Conner 3.5" hard drive data sheets, 1994, 1995.

[26] "Technology and Time-to-Market: The Two Go Hand-in-Hand", Quantum Corporation, 1995.

[27] "Basic Flaws in Internet Security and Commerce", Paul Gauthier, posting to `comp.security.unix` newsgroup, 9 October 1995, message-ID `gauthier.813274073@espresso.cs.berkeley.edu`.

[28] "cryptlib Free Encryption Library", Peter Gutmann, `http://www.cs.auckland.ac.nz/~pgut001/cryptlib.html`.

A Revocable Backup System
(extended abstract)

Dan Boneh
dabo@cs.princeton.edu

Richard J. Lipton*
rjl@cs.princeton.edu

Department of Computer Science
Princeton University
Princeton, NJ 08544

Abstract

We present a system which enables a user to remove a file from both the file system and all the backup tapes on which the file is stored. The ability to remove files from all backup tapes is desirable in many cases. Our system erases information from the backup tape without actually writing on the tape. This is achieved by applying cryptography in a new way: a block cipher is used to enable the system to "forget" information rather than protect it. Our system is easy to install and is transparent to the end user. Further, it introduces no slowdown in system performance and little slowdown in the backup procedure.

1 Introduction

On many systems the remove-file command misleads the user into thinking that his file has been permanently removed. Usually the file is still available on a backup tape. This is an important feature used to protect against accidental file erasure or system crashes. However, it has the draw back that the user is unable to completely remove his files. In many scenarios it is desirable on the part of the user to erase all copies of a certain file. This is frequently the case with personal mail messages. Similarly, a user may wish to remove the history and cache files of his web browser. Other examples include a patient removing his medical files from the hospital's system upon checkout and financial data that should be erased after a short period of time. Our goal is to make it so that the sensitive data becomes inaccessible to anyone (including the data's owner).

A naive solution is to erase the data from the file system. Then mount the backup tapes one by one and erase the sensitive data from them. This method is impractical for several reasons. First it inconveniences the user. To implement this scheme the user must call a computer operator whenever such an erasure is to take place. Furthermore, this method is quite complicated considering the backup policy which many institutes employ. Usually a backup of the entire file system is done once every time period, e.g. every month. The backup tape is then stored forever in a "cellar" or sometimes off site. This backup policy enables users to restore their entire directory structure to any point in the past. Clearly the naive approach is doomed to fail whenever this backup policy is used. The computer operator has to remove the data from many backup tapes. This procedure is painstaking, but is also insecure since the operator might "forget" to remove the data from one of the old backup tapes.

We propose a system which avoids the problems mentioned above. Our system will enable the user to remove a file from the file system and all backup tapes without ever mounting a single tape. At first this seems impossible: it is not possible to remove data from a tape without physically erasing the tape. Fortunately, cryptography enables us to do just that. The basic idea can be described as follows: when a file is backed up to tape it is first encrypted using a randomly generated key. The encrypted version of the file is the one written to tape. When the user wishes to remove the file from the backup tape he instructs the system to "forget" the key used to encrypt the file. The act of forgetting the encryption key renders the data on the tape useless. No one, including the file owner, can ever access the file again. In effect, the file has been erased from the tape. Notice that the encrypt-before-backup is completely transparent to the user. It is crucial that no one but the operating system know which key was

*Supported in part by NSF CCR–9304718.

used to encrypt the file during backup. This ensures that when the operating system is instructed to forget the encryption key, the tape data becomes unreadable.

The rest of the paper is devoted to describing an implementation of the above idea. The tricky part of the implementation is managing the encryption keys. Clearly the encryption keys have to be backed up as well. We have devised several methods for backuping the encryption keys while still preserving the fundamental properties described above.

It is interesting to compare our system to the cryptographic file system implemented by Blaze [2]. In a cryptographic file system the files are always stored in an encrypted form. Whenever a file needs to be accessed it is decrypted on the fly using the owner's key. The backup tape is a direct dump of the file system, i.e. the tape contains the encrypted version of every file. Clearly the owner of the file is the only person who has access to the backed up version of the file. This does not guarantee that his backed up files will not be accessed by an unauthorized party. The owner might be forced to reveal his key, e.g. due to a court order. Thus, the cryptographic file system does not conform to our requirements: it does not guarantee that the backup tape data will not be accessed. On the other hand, in our scheme the backed up data becomes inaccessible to everyone once it is "erased" from the tape. In addition it is important to point out that our system is very easy to install. Only the backup system has to be modified. The cryptographic file system requires far more extensive modifications.

2 Key management

In this section we give a detailed description of our backup system. In our system the user can specify a collection of files that are to be encrypted during backup. Each such file has an encryption key associated with it. The encryption key has a life time which is specified by the user. When the key expires the system generates a new key for the file and adds the expired key to a list of old keys. The maximum number of expired keys that the system remembers is a parameter specified by the user. When the list of keys is at its maximal size the oldest expired key in the list is erased to make room for the new key.

The above mechanism is very flexible. We give some examples to illustrate its use. Suppose a user is preparing a document and he wants to make sure that old and invalid copies of his draft are inaccessible to anyone. He could set the key-life for that document to be one month and instruct the system to store 12 expired keys for this document. This means that a new key is generated once a month causing a year old key to be erased. The result is that once a month all backup copies older than a year are revoked. To revoke all copies that are more than 6 month old, the user can manually instruct the system to remove all keys that expired more than 6 month ago. Of course the user can decide to delete the file altogether by instructing the system to remove all keys (including the current key) for the file. In case of disk crash the most recent version of the document can be restored from tape using the current key.

More generally, institutions may wish to adopt backup revocation policies. For instance, to prevent law suits regarding old data, an institution may decide to revoke all backup tapes that are more than 3 years old. This is done by instructing the system to delete all keys that expired more than 3 years ago. We point out once more that the revocation process has the effect of removing files from *all* old backup tapes without ever mounting a single tape.

The most important component of our system is the key management. All encryption keys are stored in one file which we refer to as the *key-file*. This file should be protected meaning that only privileged processes should be allowed to read it. The key-file contains one record per each file that has ever been encrypted during back up. There are two types of entries in the key-file: directory and file entries. The structure of a single entry in the file is described in Figure 1. The fields in a file entry contain the filename, the maximum number of expired keys that the system should store, the life time of a single key and a list of keys. The list of keys includes the current key (as the first entry in the list) followed by the expired keys ordered chronologically. When a new key is generated the keys in the list are shifted and the last (oldest) entry is lost.

The fields in a directory entry specify the directory path and indicate whether all files in the directory and subdirectories should be encrypted during backup. When the cont_flag is turned on the directory is scanned and all new files in it are added to the key-file. The `num_keys` and `key_life` fields for these new file entries are set to the values taken from the corresponding fields in the directory entry. A directory entry must always precede a file entry in order to specify the path to the file.

The key-file is permanently stored on the file system. This file is extremely important since without

it the backup tapes are useless. For this reason the key-file must be backed up as well. However, the file can not be written to tape as is. If it were written to tape the system would not be able to permanently erase keys from the key-file. This would defeat the purpose of our system. Our solution is to generate a new *master-key* during every backup. The key-file will be written to tape after it has been encrypted using this new master-key. The master-key itself is not written to tape. Notice that in case of a disk crash, the master-key is crucial for recovering the key-file. Without the master-key the key-file can not be recovered and as a result *all* backup tapes using our system become useless. For this reason the master-key must be handled with care. We discuss methods for storing the master-key in Section 2.1.

To make sure that the key-file is backed up to tape we treat it as any other file which is to be encrypted during backup. This means that the key-file is stored on the file system for which it used. Further, the key-file contains an entry which corresponds to the key-file itself. The file-name field in this entry contains the key-file name and the key field contains the master-key. Hence, the master-key is actually stored in the key-file and is used to encrypt the key-file during backup. To make sure that a new master-key is generated during every backup the key-life field is set to zero. Similarly, to make sure that only the current master-key is stored, the num-keys field is set to 1. This has the effect of revoking the old copy of the key-file during every backup.

We can now describe the details of the backup and restore operations. The schematics of the backup procedure are described in Figure 2. We briefly sketch these steps here.

Initialize When the backup process is initialized the entire key-file is loaded into memory. Then all directories in the key-file which have the `cont_flag` turned on are scanned and their contents is added to the memory image of the key file. The `touch_key_file()` routine will update the last modification date of the key-file. This guarantees that the key-file is written to tape even during an incremental backup[1].

New keys Next, new keys are generated for all files who need them. A new key is generated for all files for which `date_key_issued + key_life < current_date`. The `date_key_issued` field is updated for each new generated key.

Write all files to tape Loop on all files in the file system. Each file is dumped to tape, after encryption if necessary. The old master-key is erased from the key-file just before the key-file itself is dumped to tape. This ensures that the old master-key is *not* written to tape when the key-file is backed up. Had this not been done, old master-keys could be read from the backup tape. This would be disastrous for our system.

Terminate Finally, write the key-file back to disk and store the new master-key using the methods described in Section 2.1.

The modifications to the restore operation are even simpler. During restore we use the key-file to find the appropriate key for each file. The system first reads the date written on the backup tape. This date is the date on which the backup was done. For each file to be restored the system retrieves the key used to encrypt the file on the backup date. This key is then used to decrypt the file when it is read from the tape.

In case of a disk crash the first thing that has to be recovered is the key-file. This is done by creating a temporary key-file containing a single entry. This entry contains the key-file as its file name and the master-key as the key. The operator can now use restore to recover and decrypt the most recent backed up version of the key-file. Now a full restore of the file system is possible.

2.1 Master key management

As was mentioned above, the master-key is a crucial component of the system. Without it, all the backup tapes are useless. We first describe the properties that a master-key storage system must satisfy. Observe that the master-key can not be written to tape in the clear. If it was, then the key-file on the tape would be accessible and with it all the files on the tape. This will again defeat the purpose of our system. Furthermore, wherever we choose to store the master-keys we must make sure that only the most recent key is accessible. As before, an old master-key will enable access to the contents of old tapes. Hence, any system which stores master keys must "forget" all but the most recent one.

We propose two solutions which can be used in conjunction with one another. The first one is the simplest; at the end of each backup the computer operator is asked to write down (on paper or on a floppy disk) the current master-key. He then destroys his copy of the previous master-key.

The second method involves an internet server

[1] An incremental backup of a certain level only backups files that were modified since the last backup of that level.

```
union keyfile_entry {

  struct file_entry {              /* Structure of file entry        */
    char *filename;
    char *keys[];                  /* Keys used to encrypt file      */
    int  num_keys;                 /* Number of keys to be saved     */
    time_t key_life;               /* Life time of a single key      */
    time_t date_key_issued;        /* Date current encryption key    */
                                   /*    for file was issued         */
    time_t last_backup_date;       /* Date file was last backed up   */
  } file

  struct dir_entry {               /* Structure of directory entry   */
    char *dirpath;                 /* Directory path                 */
    char cont_flag;                /* Indicate if files in dir       */
                                   /*   should be added to key-file  */
    int  num_keys;                 /* Default num keys to be saved   */
    time_t key_life;               /* Default life time of a key     */
  } dir;
}
```

Figure 1: key-file structure

```
load_key_file();
scan_sub_dir();
generate_new_keys();
touch_key_file();                              /* key-file will be stored    */
                                               /* during incremental backup */
Loop on all files in file-system {

  file_entry = get_key_file_entry(current_file);
  if (file_entry != NULL) {                    /* If file has an entry in    */
    key = get_key(file_entry);                 /* key-file then get          */
  else                                         /* encryption key. Otherwise, */
    key = NULL;                                /* don't encrypt file.        */

  if (is_key_file(current_file))               /* Erase current master-key   */
    erase_master_key_from_disk();              /* before writing key-file.   */

  Loop on blocks in file {                     /* Write file blocks to tape  */
                                               /* encrypting them if         */
    if (key != NULL)                           /* necessary.                 */
      encrypt(current_block, key);

    dump(current_block);                       /* Write block to tape.       */
  }
  if (file_entry != NULL)                      /* Update backup date field   */
    file_entry->last_backup_date = current_date;
}

write_key_file();      /* Write key-file back to disk.                       */
store_master_key();    /* Take extra care to store the new master key.       */
```

Figure 2: Backup operation

which is used for master-key storage. Suppose a site performs daily incremental backups. This is common in many sites. The master-key-storage-server generates a public/private key pair on a daily basis. Every day all sites performing backups encrypt their daily master-key using the server's current public-key and write the resulting string on their backup tape. In case of a disk crash a site can recover its daily master-key by performing the following steps: the local computer operator first reads the encryption of the master-key written on his tape. He then sends the encrypted master-key to the storage-server. The storage-server, after authenticating the identity of the sender, decrypts the master-key and sends the result back to the computer operator. The operator can now restore his file system. The same public/private key pair can be used by all sites in the world which use daily (incremental) backups. Hence, the master-key-storage-server simply provides the service of generating a public/private key pair on a daily basis.

The above description is not much different than a standard key escrow system. The new twist is that to make sure that old master-keys are inaccessible the master-key-storage-server must erase all but its most recent private-key. Otherwise old master-keys can be recovered from old tapes. These master-keys then enable anyone to read the contents of old tapes. Since the storage-server is providing a commercial service it is in its best interest to be trustworthy and indeed "forget" all old private keys. To increase the security and reliability of the scheme one can use k out of l secret sharing techniques [4]. This means that a given site will employ l storage-servers and exactly k of them are required to recover the daily master-key. Now an old master-key can not be recovered even if $k-1$ of the storage-severs are untrustworthy. Similarly, even if $l-k$ storage-servers crash and lose their current private-key a site can still recover its current master-key. The parameters l and k can be fine tuned to the site's needs.

For increased reliability some sites may wish to be able to access a small number (e.g. three) of old master-keys. For instance, if the most recent backup tape is lost, the previous one can be used if the corresponding master-key is still accessible. As a result some sites may want the storage-server to save a small number of its most recent private-keys. To accommodate this need the storage server can offer k (where $k < 10$) types of public/private key pairs. For type 1 only the most recent private key is available. For type 2 the two most recent private keys are available, etc. This arrangement requires the storage-server to generate k new public/private key pairs daily. A site who wishes to have access to its three most recent master-keys may use a type 3 public key published by the storage server.

3 The user interface

Two utilities enable a user to interact with our system. The first utility enables a user to declare that a file or directory has the revocable-backup attribute. The second enables a user to revoke backup copies of his files. These two utilities comprise the most basic interface. Naturally utilities for obtaining the status of a given file are also provided.

mkrvcbl The make-revocable command will add a file (or directory) to the key-file. The user can specify the key-life and num-keys parameters on the command line. The default values for these parameters is infinity for the key-life and one for num-keys. This means that there is a single key associated with the file throughout the life of the file. This enables the user to revoke all backup copies of the file when he wishes to completely remove the file from the system. Executing `mkrvcbl dir-name` will add the directory to the key-file with the `cont_flag` turned on. During backup all files in the directory and its subdirectories will be added to the key-file. Only the owner of the file (or directory) is permitted to apply `mkrvcbl` to the file (or directory).

rvkbck The revoke-backup command is activated as `rvkbck file-name [date]`. Only the file owner is permitted to apply `rvkbck` to a file. The command removes all keys that expired before `date` from the file's entry in the key-file. If the `date` parameter is left blank the entire file's record is removed from the key-file. In doing so, all keys used to decrypt the file from the backup tapes are lost. As a result, the file can no longer be restored from the backup tapes. Unfortunately this is not quite true. Recall that the key-file itself is also backed up on tape. Hence, the removed entry can still be found in the backup version of the key-file. However, at the next backup the master-key will change making the old backed up version of the key-file useless. Therefore, the entry is permanently removed from the key-file after the next backup operation. For example, in a system where an incremental backup takes place daily the `rvkbck file-name` command will permanently remove the file within 24 hours of the time the command is issued.

The two utilities described above enable a user to

manipulate the key-file according to his needs. By definition, the key-file contains an entry for every file that has ever been encrypted during backup (and has not been revoked). This could make the key-file large. Once every time period, e.g. once a year, the computer operator may wish to prune the key-file by removing all entries which correspond to files which are no longer present on the file system. This is done using the *prune-key-file* utility.

prunekeyfile The command `prunekeyfile date` will remove all entries in the key-file which correspond to files which no longer exist on the file system and whose last backup date is prior to 'date'. The utility will write the removed entries to a file called `key-file.'date'`. This file will be backed up to tape as a regular file (without encryption) and then removed from the file system. Notice that by doing so the user is no longer able to revoke the backed up copy of the files corresponding to the removed entries. This is fine since these files are old files which are no longer present on the file system.

4 Summary and future work

We described a system which enables a user to permanently remove a file from the file system and all backup tapes. The ability to revoke backup copies of files is important and may be of interest to many institutions. Our system is very easy to install and provides a simple user interface.

Our scheme applies cryptography in a new way. Here cryptography is used to erase information rather than protect it. Since the backed up files are stored for extended periods of time it is desirable that the block cipher used to encrypt the files be extremely secure. Hence we suggest using block ciphers with longer keys than are usually used. For instance one could apply triple DES twice to obtain a 224 bit key.

To simplify the installation of our system we chose not to modify the existing file system. Our implementation has the drawback that the backup key information does not become a part of the file. Thus, when the file is moved or copied the resulting file will not be securely backed up. A full-scale implementation of our scheme could embed a new file attribute in the file header. This attribute would indicate that the file is to be securely backed up. This way when the file is moved or copied, the new file will have the same attributes.

The size of the key-file can be reduced by incorporating the ideas used in Lamport's one-time password scheme [3, pp. 230–232]. We thank Steven Bellovin [1] for pointing this out. Let f be a one way permutation. The idea is to only store the oldest accessible key in the key-file. The more recent keys can be obtained by repeatedly applying f to this key. When the oldest key k expires it is simply replaced by $f(k)$. Since given $f(k)$ it is hard to compute k we may say that k has been "forgotten". For efficiency it is desirable to store both the oldest and most recent keys in the key-file. Using this approach we can make do with storing only two keys per key-file entry.

As a final note we point out that standard UNIX backup utilities, e.g. dump and tar, do not enable a user to specify a collection of files that should not be backed up. There are several types of files for which this option is important due to privacy considerations. The typical example is the history and cache files of the user's web browser. On the one hand, the information is usually not important to the user. On the other hand the information might be private and the user may not want extra copies of it floating around. We suggest that this feature be incorporated into commercial backup systems.

Acknowledgments

We thank Jim Roberts and Matt Norcross for a very helpful explanation of our local backup system.

References

[1] S. Bellovin, private communications.

[2] M. Blaze, "A Cryptographic File System for Unix",
available at `http://www.cert-kr.or.kr/doc/Crypto-File-System.ps.asc.html`

[3] C. Kaufman, R. Perlman, M. Speciner, "Network Security", Prentice Hall, 1995.

[4] A. Shamir, "How to share a secret", CACM, Vol. 22, Nov. 1979, pp. 612–613.

Building Blocks for Atomicity in Electronic Commerce

Jiawen Su J. D. Tygar
sjw@cs.cmu.edu tygar@cs.cmu.edu
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Atomicity is clearly a central problem for electronic commerce protocols — we can not tolerate electronic commerce systems where money is arbitrarily created or destroyed. Moreover, these atomicity properties should be retained in the event of component failures in distributed systems. In this paper, we enumerate several classes of atomic protocols. We then give two fundamental building blocks for building atomic electronic commerce protocols: encryption-based atomicity and authority-based atomicity. We then illustrate these building blocks by considering variations of payment-server based protocols that use these different building blocks. The results give a contrast to the class of protocols that we have previously examined in our work with NetBill.

1 Introduction

We study electronic commerce protocols that allow users or agents to transfer funds on a network. In these electronic commerce protocols, a customer C makes a purchase by transferring funds to a merchant M.

What if communications or some other subsystem fails during the transfer? It is reasonable to require that the resulting electronic commerce system impose an *atomicity* condition: either the funds transfer should complete (M has the money; C does not) or it should not occur at all (C has the money; M does not). It is unacceptable if the protocol leaves us in a partial or ambiguous state: for example, if neither M nor C has the money, then our system has destroyed value. Similarly, if both M and C have the money, or even if both believe they have the money, chaos or counterfeiting can result.

In [18], the second author argued that this is an example of *money atomicity*: money should not be created nor destroyed by an electronic commerce protocol. The second author then defined two stronger classes of atomicity: *goods atomicity* (transfer of goods occurs if and only if the money transfer happens and certified delivery (all parties can prove the exact content of goods transferred after the fact with a money atomic protocol.) [18] has an extensive discussion of these different types of atomicity, and shows examples of protocols meeting and not meeting these atomicity concerns.

A close examination of published electronic commerce protocols quickly reveals a number that do not satisfy atomicity constraints. These protocols appear to use very strong cryptographic methods, but they fail to consider the possibility of communication failure.

Many of the "electronic currency" protocols (including [3, 8, 13]) exemplify the difficulties of non-atomic systems. Consider the policy for handling communications failures. Consider the policy for handling communications failure as a customer is transferring funds to a merchant: the merchant may or may not have received the funds, but the customer does not know the status of the funds transfer. If the policy is to deny the customer access to the funds that have started to transfer (but may have not been received by the merchant), then this will sometimes result in lost money. On the other hand, if the policy is to allow the customer access to the funds, then the customer and the merchant may both believe that they have access to the same funds. This can result in double-spending. In systems such as these, that will be indicated as a potential fraud, and will result in the customer being charged with counterfeiting — even though the customer may have acted in good faith according to the policy of the system.

Atomicity is not a new concern; it has been considered for years in the transaction processing environment. There

We gratefully acknowledge the support of the National Science Foundation (under cooperative agreement IRI-9411299), the US Postal Service, and Visa International for the support of this research. This paper is solely the opinion of the authors and does not necessarily reflect the opinion of the funding agencies.

are some excellent references on general techniques for achieve atomicity in electronic commerce protocols ([12, 11] are outstanding surveys of the field.) This paper is a first attempt to enumerate some possible building blocks for generating atomic protocols in electronic commerce.

Our concern with these topics is not purely academic. At CMU, we are currently building a system called NetBill. NetBill has a protocol (see [10, 16]) optimized for highly atomic electronic transactions. This paper considers several variations on those protocols which have radically different performance considerations.

We limit our examples in this paper to payment servers; but the second author has recently worked with others to develop a number of atomic versions of electronic cash [4, 5] and the Mastercard/VIS SET Secure Electronic Transaction [15] secure protocol for transmitting credit cards over the network [4, 6].

2 Payment models

Most proposed electronic payment systems fall into three broad categories: electronic currency, debit/credit instrument and secure credit card.

In an electronic currency system, money is denoted as a *token* which is a sequence of digital bits. Most electronic currency systems strive to model traditional currency: they attempt to provide anonymous and unforgeable tokens. However, it is especially easy to copy a byte string, so these protocols use special techniques to prevent double-spending tokens because almost no other things are as easily copyable as digital bits. Chaum pioneered electronic currency protocols [8]; there is a very wide literature on systems derived from Chaum's framework. In Chaum's system, tokens are withdrawn from an issuing bank. To prevent double-spending a token, each token includes hidden account information. If a customer double spends a token, their is sufficient information to uniquely identify the customer.

In debit/credit instrument systems, customers and merchants register accounts with payment servers. When a customer buys goods from a merchant, he or she signs a payment instrument directed to his payment server. Some examples of debit/credit instrument systems are NetBill [16] and NetCheque [14].

Secure credit card systems base the payment on traditional credit cards [2, 15]. A customer's credit card number is securely transferred to a merchant in case of payment. Then the merchant processes the payment through a traditional credit card transaction. These protocols often add a twist: they attempt to protect the merchant from obtaining the customer's credit card number. Instead, the transaction is processed only by the acquiring bank of the merchant; preventing a wide class of credit card fraud: merchant fraud where the merchant misuses the customer's credit card number.

3 Security considerations

There are many cryptographic and security considerations that must be satisfied to make a safe, correct, and secure electronic commerce protocol. To make our discussion more focused, we do not consider these issues in the protocols discussed below. In particular **the protocols below in sections 4 and 5 should not be considered to be secure for actual use.** In fact, integrating security concerns with atomicity is a non-trivial problem and is illustrated in detail in [10, 18].

In particular, here is a (not exhaustive) list of some considerations that apply (some excellent references on further essential elements for secure protocols is contained in [1, 17]):

- **Privacy:** Communications must be protected from eavesdropping from an unintended party. A private communications channel, or a virtual private communications channel, should generally be used in the protocol. There are several ways that this could be achieved: a physically secure line might be used, a shared symmetric key cryptosystem could be exploited, messages could be encrypted in the public key of the intended recipient, etc. These various techniques lead to radically different performance considerations: for example, the costs of key exchange and encryption can change the cost equation making some protocols feasible and other infeasible.

- **Nonrepudiation:** For showing some aspects of certified delivery, it may be necessary to prove to a third party that a certain communication was really received from another party. For example, it is certainly useful for a receipt to be nonrepudiable, so that the holder of a receipt can use it as proof in case of a dispute over a transaction. One way to achieve this is through the use of digital signatures; however these can add substantially to the running time of the protocol.

- **Protection against replay:** One of the most common attacks on protocols involves replaying certain messages. Anyone who has been double-billed on his credit card for a single purchase is familiar with

the consequences of this attack. To address this, we use the techniques of *idempotence* (discussed in section 4 and *nonces*. Nonces are unique identifiers that associate a set of messages uniquely with a single transaction. Unfortunately, the generation of nonces and the protection against their malicious use is non-trivial. For example, if an opponent can guess a valid nonce value, he may be able to insert the message in a way that confuses the analysis.

By putting the issues of privacy, nonrepudiation, and protection against to the side for a moment, we can focus on the vital issues of atomicity. We do not consider here the full integration of these elements.

4 Atomic building blocks

We consider protocols that preserve atomicity. These protocols must maintain an all-or-nothing property for transfer of funds (and digital goods) — either the transaction completes or the effect is restored as if the transaction did not occur. We give two building blocks: encryption-based atomicity and authority-based atomicity.

Now, in the building blocks below, we have several parties defined: *S*: A sender who attempts to send some data *m* (such as digital goods, a value token, a credit card number, etc) to a recipient; *R*: A receiver who receives the data; and *A*: A trusted third party recipient of the data. We can assume that *S* and *R* may try to deviate from the protocol, but we assume that *A* will always comply with the protocol requirements. (In fact, in more sophisticated analyses, such as [10, 18], we can weaken the assumption and consider the possibility of a corrupt third party; but here we do not consider that case.)

We use the notation $\{m\}_k$ to denote that message *m* is encrypted under key *k* (for example, under a symmetric key cryptosystem such as DES.) We use TID to denote a unique and unguessable (see section 3) transaction ID. And we use the notation sig *x* to denote *x* in a digitally signed format.

Finally, we note that all of our messages are designed to be *idempotent*, they can be repeated more than once with no additional effect. For example, a message that says, "decrease my account by one unit", is not idempotent since its repetition will result in the account being decremented by more than just one unit. However, a message that is uniquely identified by a transaction ID can be made idempotent since the receiver of the message can ignore repeated copies of the same message tagged with the same transaction ID. (See [11] for more on the subject of idempotence.)

4.1 Encryption-based atomicity

Our first building block prevents data (such as a token or digital goods) from being read by the recipient before a key can be sent to a trusted third party. This third party can then arbitrate to ensure atomic delivery.[1]

1. S → R: $\{m\}_k$, TID
2. R → S: sig $\{m\}_k$, TID
3. S → A: k, TID
4. S → R: k, TID

Note that if communications fail before or during step 3, the transaction will not take place. On the other hand, if communications fail after step 3, *R* may fail to receive the decryption key *k*. However, if we trust the authority *A* to freely give *k* on request, requests it, then *S* and *R* can always complete the transaction even if they (or their communication links) fail after step 3.

Message 2 provides *S* with a receipt of the fact that *R* received message 1 intact. Later, if their is a dispute over the contents of the message, this signature, together with a signed copy of *k* obtained from *A*, can be used to prove the contents of *m* to any third party (such as a digital judge!)

4.2 Authority-based atomicity

A different approach to atomicity can be to have the trusted authority *A* not only hold cryptographic keys but also the messages themselves in escrow. This way, *A* becomes a trusted communication agent. This can result in fairly expensive storage costs for *A*. Here is a simple authority-based atomic transfer of message:

1. S → A: *m*, R, TID
2. A → R: "message available", TID
3. R → A: "send me the message", TID
4. A → R: *m*, TID

Here, *A* will store message *m* and continue to transmit message 2 until *R* finally picks up his message.

5 Atomic protocols

In this section, we analyze various electronic commerce models and apply our building blocks to achieve atomicity for debit/credit instruments.

[1] The careful reader will note that this protocol and the ones that follow do not provide privacy, nonrepudiation, or protection against replay attacks. We have distilled these elements out of this set of building blocks. In particular, because this protocol and the ones that follow are not designed to be secure, we do not recommend that they be used without modification. See section 3 for more discussion of these issues.

(We discuss one of the three major electronic commerce models below. What about the other two models electronic currency and secure credit card information? Electronic currency presents special challenges for atomicity because it required us to provide anonymous and atomic transactions. The integration of these two elements is extremely complicated; recently Jean Camp, Mike Harkavy, Bennet Yee, and the second author have developed a family of protocols to achieve this, and a description of that protocol is available in [4, 5]. For secure credit card transactions, Jean Camp, Marvin Sirbu, and the second author [4, 6] have recently shown how to integrate certified delivery with the new SET protocol [15].)

5.1 Debit/Credit Instrument

In a debit/credit model, both customers C and merchants M register accounts with a payment server P. (P is assumed to act as a trusted third-party authority, and to abide by the conditions discussed above.) The payment server is assumed to use a locally atomic database to register transfer of funds among accounts, so money atomicity follows trivially. However, the problem of goods atomicity (money exchanged for goods) and certified delivery (proof of the contents of items delivered) is non-trivial.

We assume the case that the items being purchased are digital goods to illustrate the atomicity concerns.

In [10, 18] we consider this problem at length. Here, we merely summarize a few alternatives.

Case 1: Without atomicity

1. C → M: price-inquiry, TID
2. M → C: price, TID
3. C → M: instrument, TID
4. M → P: instrument, TID
5. P → M: status of payment, TID
6. M → C: item, TID

Comments. Here the item is the digital goods to be sold. The example above is clearly not goods atomic, since the merchant could decline to send message 6. Note that the instrument in this case could be an invoice, and that in steps 3 and 4, both M and C have an opportunity to agree on the price of the item. In most cases, the instrument should be signed by C and include the TID and an item description to ensure that the instrument originated in a nonrepudiable fashion from C and is not being replayed. (Note that this is not a secure protocol; see section 3 or [10, 18] for more details.) They payment is handled online by the payment server, which then must report the success or failure of the transfer to the customer.

Case 2: Encryption-based atomicity

1. C → M: price-inquiry, TID
2. M → C: price, $\{item\}_k$, TID
3. C → M: instrument, sig $\{item\}_k$, TID
4. M → P: instrument, k, sig $\{item\}_k$, TID
5. P → M: status of payment, TID
6. M → C: (if successful) k, TID

Comments. Here, we have the item send encrypted under k in step 2. C then prepares a signed copy of the encrypted item, which in step 4 should be counter-signed by the merchant, so both parties agree on the contents of the item. The instrument should contain information such as the TID and a description of the item for maximum protection. When the counter-signed item, together with the decryption key k and the instrument, are registered at P, it enables P to give proof of certified delivery — contents are registered at P. If steps 5 or 6 fail, then M and C can directly query P to discover the status of the transaction.

Note that instead of a full signature in steps 3 and 4, it is desirable to take signatures of cryptographic checksums (such as MD5) of the data. This not only reduces storage costs at P, but it provides additional privacy against P reading the contents of the item sent from M to C.

Finally, note that this protocol is only an example of the application of two-phase commitment [11, 12] to the problem of electronic commerce. Two-phase commitment is a well known technique for achieving atomicity.

Case 3: Authority-based atomicity

1. C → M: price-inquiry, TID
2. M → C: price, TID
3. C → M: instrument, TID
4. M → P: instrument, item, TID
5. P → M: status of payment, TID
6. P → C: (if successful) item, TID

Comments. Authority-based atomicity requires the payment server to retain much more information than it would under encryption-based atomicity. In particular, it must retain the full contents of the item indefinitely. Certified delivery is satisfied by the escrowed copy of the item stored at the payment server. Since messages 5 and 6 may not be received, the payment server must respond to queries from either M or C on the status of the payment.

In addition to the storage required in this protocol, there is also a very serious difficulty in that the payment server can easily read the contents of the item. Now, it may be possible to solve this problem through the use of public

key cryptography: M could encrypt the item in C's public key before transmitting it to P. However, to verify the contents of the message, it would be necessary for C to disclose his private key to a third party (such as a digital judge.) This would have the unfortunately drawback of disclosing all of C's items received encrypted under his public key to the third party. A better solution in this case would use a one-time only public-private key pair. Safely handling the key management information in this case would add substantial complexity to the protocol.

6 Open Problems

This work leaves a number of important issues open including:

- lower performance bounds on atomic protocols;
- the performance impact of integrating privacy, non-repudiation, and protection against replay attack into atomic protocols;
- techniques to prove atomicity (recent unpublished work of the second author with Nevin Heintze, Jeannette Wing, and H. C. Wong gives some preliminary indications that model checking may be an especially fruitful technique to attack the problem of checking atomicity of protocols); and
- removing blocking (our protocols are all derived from two-phase commitment, and thus share the blocking characteristics of that protocol — there are a variety of so-called "non-blocking" commitment protocols that have been studied and some of these may yield important results in the electronic commmerce sphere [11, 12].)

For a more comprehensive list of open problems, see [18].

7 Acknowledgements

Discussions with the following people have illuminated our understanding of electronic commerce atomicity concerns: Jean Camp, Ben Cox, Nevin Heintze, Mike Harkavy, Cliff Neuman, Marvin Sirbu, and Bennet Yee.

References

[1] M. Abadi and R.M. Needham. Prudent Engineering Practice for Cryptographic Protocols. *Proceedings of the IEEE Symposium on Research in Security and Privacy*, May 1994. pages 122-136.

[2] M. Bellare *et al.* iKP — A Family of Secure Electronic Payment Protocols. In *Proceedings of the First USENIX Workshop on Electronic Commerce*, July 1995.

[3] S. Brands. Untraceable Off-line Cash in Wallet with Observers. *Advances in Cryptology-Crypto'93*, 1994.

[4] J. Camp. *Privacy and Electronic Commerce.* PhD thesis, Engineering and Public Policy Department, Carnegie Mellon University. To appear.

[5] J. Camp, M. Harkavy, J. D. Tygar, and B. Yee. *Atomic Electronic Cash.* To appear.

[6] J. Camp, M. Sirbu, J. D. Tygar. *Certified Delivery with SET.* To appear.

[7] D. Chaum. Security without Identification: Transaction Systems to Make Big Brothers Obsolete. *Communications of ACM*, 28(10), 1985. pages 1030-1044.

[8] D. Chaum, A. Fiat, and M. Naor. Untraceable Electronic Cash. In *Advances in Cryptology-Crypto'88*, 1989. pages 644-654

[9] D. Chaum and T.P. Pedersen. Wallet Databases with Observers. *Advances in Cryptology-Crypto'92*, 1993.

[10] B. Cox, M. Sirbu, and J. D. Tygar. NetBill Security and Transaction Protocol. In *Proceedings of the First USENIX Workshop on Electronic Commerce*, July 1995, pages 77-88.

[11] J. Gray and A. Reuter. *Transaction Processing: Techniques and Concepts.* Morgan Kaufmann, San Mateo, CA, 1994.

[12] N. Lynch, M. Merrit, W. Weihl, A. Fekete. *Atomic Transactions.* Morgan Kaufmann, San Mateo, CA 1994.

[13] T. Okamoto and K. Ohta. Universal Electronic Cash. *Advances in Cryptology-Crypto'91*, 1992.

[14] B. Neuman, G. Medvinsky. Requirements for Network Payment: The NetCheque Perspective. *Proceedings of IEEE Compcom'95*, March 1995.

[15] Secure Electronic Transactions Specification. This information is available from the WWW pages of both Mastercard (www.mastercard.com) and Visa (www.visa.com) home pages; for example, see http://www.mastercard.com/set/set.htm.

[16] M. Sirbu and J. D. Tygar. NetBill: An Internet Commerce System Optimized for Network Delivered Services, *IEEE Personal Communications*, August 1995. pages 6-11.

[17] P. Syverson. Limitations on Design Principles for Public Key Protocols. In *Proceedings of 1996 IEEE Symposium on Security and Privacy*, pages 62-72.

[18] J. D. Tygar. Atomicity in Electronic Commerce, *ACM/IEEE Conference on Principles of Distributed Computation*, 1995, pages 8-26.

Here are some relevant URLs for the paper:

NetBill site: http://www.ini.cmu.edu/NETBILL/home.html Su's home page: http://www.cs.cmu.edu/afs/cs/usr/sjw/www/sjw.html Tygar's home page: http://www.cs.cmu.edu/afs/cs/usr/tygar/www/home.html

Kerberos on Wall Street

Isaac Hollander
hollande@ms.com

P. Rajaram
rajaram@ms.com

Constantin Tanno
tannoc@ms.com

Morgan Stanley & Co. Incorporated
750 Seventh Avenue
New York, New York 10019

ABSTRACT

Morgan Stanley, an international financial services firm, has a significant investment in Kerberos. We have been using Kerberos 4 since 1993, and began a transition to Kerberos 5 in 1995. Kerberos has helped us progress towards solving the classic security problems of cleartext passwords and single sign-on. However, because of Morgan Stanley's specialized requirements, and a lack of support for Kerberos among operating system and application vendors, we have faced a number of practical integration problems not faced by the research community. This paper discusses those problems, the tools we have built to solve them, and the areas in which we feel vendors must provide better support for Kerberos.

1. Introduction

Morgan Stanley is an international financial services firm with 37 offices in 22 countries. Since 1993, Morgan Stanley has used Kerberos for initial login authentication by 3500 UNIX accounts (75% of total) on 4000 UNIX machines (80% of total) spread over a large WAN over 20 sites on 3 continents. Several Kerberized applications were also deployed: those originally supplied by MIT (rsh/rlogin/rcp), as well as a handful of proprietary Morgan Stanley Kerberized client-server applications. Kerberos has solved the classic security problems caused by cleartext passwords traveling over networks, and has spurred development of authenticated client-server business applications. It has also helped us take a big step towards providing single sign-on access to all computing platforms in the Firm.

While the theory and implementation of the Kerberos protocol is of great interest to corporate Information Technology (IT) departments, a host of practical integration problems exist that were not generally faced by its designers and implements. This paper discusses the use of Kerberos at Morgan Stanley, some of the problems encountered during Kerberos 5 deployment and solutions we chose.

2. Transitioning to Kerberos 5

Our initial Kerberos deployment was based on the stock Version 4 distribution from MIT, to which we added functionality such as incremental hierarchical database propagation and a rudimentary authorization server. Unfortunately, some of the added functionality was incompatible with the basic Kerberos 4 protocol and some of the Kerberized applications distributed by MIT. Because of these incompatibilities, well-known limitations of the Kerberos 4 protocol [Bellovin & Merrit, 1990], as well as the ongoing cost of supporting in-house code, Morgan Stanley decided in 1995 to deploy to a Kerberos 5 implementation purchased from an outside vendor. The aim of the Kerberos 5 deployment was to replace the current Kerberos 4 infrastructure, extend Kerberos to all platforms used by the Firm, including PCs, and leverage the new Kerberos 5 functionality.

2.1 Transition Planning

Deploying any software into corporate production environments requires careful planning, thorough testing, and coordination with many groups in IT and other business units. New software must be rolled out in a series of small, reversible steps to minimize the impact on the production environment in case of unforeseen problems. In addition to these requirements, we had to meet the following goals to effect a smooth transition to Kerberos 5:

1. Backwards compatibility: Existing production software should continue to work unchanged to the greatest extent possible.
2. Transparency: A business user should not need to know or care if a system is a Kerberos 4 or a Kerberos 5 client.

2.2 Deployment Strategy

Kerberos 4 and Kerberos 5 servers coexist during the transition period. The contents of the databases are kept synchronized using a database propagation gateway described later. Therefore, all principals have equivalent entries on both servers. As each set of client machines is upgraded to Kerberos 5, the krb.conf file is updated to point to a Kerberos 5 server.

Our deployment strategy for Kerberos 5 in Morgan Stanley's UNIX environments consisted of two parallel tracks:

1. For machines which had never been Kerberized, we proceeded on a site-by-site basis to convert machines and modify individual user /etc/passwd entries.
2. Conversion of machines fully integrated into our Kerberos 4 environment was substantially more complex. Batch (cron) jobs requiring Kerberos credentials required minor, but necessary modifications. We chose to convert desktop workstations first, on a site-by-site basis, since batch processing occurs primarily on servers. Servers were converted only after verification of batch processing by responsible development teams.

Great care was taken to ensure that programs like rsh worked seamlessly between hosts that had been converted to Kerberos 5 and those that had not yet been converted. Most of our production applications depend on rsh.

3. User Transparency

To make the transition to Kerberos as transparent as possible for users on both Kerberized and non-Kerberized machines, we enhanced some standard programs and developed some new tools. We were also forced to wrap many standard programs, such as the Berkeley rsh client, to provide different Kerberos related behavior via the same file system path.

3.1. Integrated Login

In many Kerberos environments, users enter two passwords: one to authenticate with a standard crypt() based login, and another to obtain Kerberos credentials via kinit. One of the enhancements added by Morgan Stanley to Kerberos 4 was to integrate Kerberos into login. Login checks the second field of a user's /etc/passwd entry; if it contains a '*' in place of an encrypted password, the user is Kerberized and the password is authenticated against Kerberos. This supplies the user with valid Kerberos credentials for the session.

The deployment of Kerberos 5 required additional modifications to login to obtain Kerberos 5 credentials. However, our login still obtains Kerberos 4 credentials for backward compatibility, as we expect to have Kerberos 4 dependent applications for the foreseeable future.

We use the MIT krb524 package to allow login to convert Kerberos 5 tickets to Kerberos 4 tickets. Our vendor's Kerberos 5 KDC also services requests for Kerberos 4 authentication. However, production applications have come to rely upon non-standard, legacy modifications made to our in-house Kerberos 4 KDC to provide support for long-lived tickets and multi-homed hosts. We therefore could not use our vendor's KDC for Kerberos 4 authentication in login directly, and decided to use the krb524 package instead.

Login now also obtains AFS (Andrew File System) credentials for the user. Login requests a Kerberos 5 AFS service ticket, which is converted to version 4 service ticket using krb524 and then transformed into AFS kernel tokens. (The Kerberos 5 AFS ticket cannot be utilized directly because AFS uses the Kerberos 4 protocol.) This feature is vital because Morgan Stanley relies heavily upon AFS; AFS contains most user home directories as well as major firm-wide software distributions [Gittler et al. 1995].

We similarly modified other basic system entry points, such as in.ftpd, in.rexecd, xlock, and xdm. The result is that typical users never have to use kinit to obtain or refresh their credentials on their desktop workstations.

Though Kerberos distributions now include integrated login programs, we chose to base our modifications on operating system vendor source code. Though additional support costs were incurred, we were not comfortable with semantic differences between the

operating system vendor's programs and the Kerberos vendor's replacement programs.

3.2. Ak5log

In addition to integrating AFS authentication into system entry points, we implemented a standalone AFS authentication program, ak5log, which obtains AFS credentials with a user's TGT. ak5log is used to obtain AFS credentials when a user forwards tickets to a remote system, as is possible with the Kerberos 5 versions of rsh, rlogin, and telnet.

We modeled ak5log after the publicly available Kerberos 4 based aklog program and patches suggested by Engert [Engert 1994; Engert 1995]. Our implementation, however, is somewhat different. The AFS authentication code is isolated into a library used by ak5log, login, and other programs that authenticate users. In addition, ak5log by default obtains tokens for every cell known to the kernel. This is important to preserve transparency for business users, who frequently access files in different cells.

The issue of credential lifetimes is problematic when integrating Kerberos 5, Kerberos 4 and AFS logins. For both Kerberos 4 and AFS, the ticket lifetime is contained in one byte; However, this byte is interpreted differently. Under Kerberos 4, each increment of the lifetime byte corresponds to 5 minutes. Under AFS, lifetimes from 0 to 128 are interpreted as 5 minute increments, but lifetimes between 129 and 191 correspond to fixed values in a table in the AFS libraries that are roughly exponentially increasing. A lifetime of 191 (0xBF) corresponds to 1 month.

Because of these differing interpretations of credential lifetimes, the lifetime of a Kerberos 4 AFS service ticket and the length of time the AFS server will allow file system access do not precisely correspond. Although Kerberos 5 tickets support longer lifetimes, this problem is still extant in ak5log because the Kerberos 5 AFS service ticket must be converted to a Kerberos 4 AFS service ticket via krb524. We avoid the problem by simply obtaining AFS tokens valid for a month, thus ensuring that business users will never be denied access to the AFS file system during their login session.

3.3. Trojan Horse Screen Lock

A goal of the Kerberos 5 deployment was to Kerberize the remaining 25% of the Firm's UNIX accounts. These accounts had been administered by a separate technology department within Morgan Stanley prior to 1994. They had not deployed Kerberos 4 and thus were still using the crypt() based login supplied with their operating system.

In order to conveniently convert these accounts, we implemented a special method for Kerberizing users en masse. The standard method for setting Kerberos passwords for new users - assigning the user a known password which they are forced to change immediately upon login - was not feasible due to inconvenience to users and support staff.

The solution we chose was a special version of our screen lock program that would add a new principal to the Kerberos database when an non-Kerberized user unlocked their screen. The new principal's password would be set to the password the user had just entered. An admin password encoded into the binary was used to authenticate to the kadmind server in order to effect the addition. Syslog messages generated during such an event were used to notify sysadmin staff to update the user's /etc/passwd entry (i.e. replace the encrypted password with a '*').

Such programs present security risks. Users who reverse-engineer the admin password out of the executable program could effectively gain access to any UNIX account in the Firm. To reduce this danger, the admin password is not present in the executable in cleartext; rather, it is obscured by encrypting it with a known key. In addition, the program is deployed for a very short time. These measures, and the fact that most Morgan Stanley users lack the technical expertise to 'crack' this simple encryption scheme, made the Trojan horse screen lock approach an acceptable compromise between security and practicality.

3.4. Incremental Gateway

In worldwide production environments, it is unacceptable to force a business user to have to wait until the next batch database dump before using a new Kerberized account. Therefore, Morgan Stanley added a hierarchical incremental Kerberos database propagation mechanism to Kerberos 4. This allowed multiple redundant Kerberos servers, to each have a continuously synchronized database. It also obviated the need for batch database replication across WAN links, freeing up bandwidth for vital market data flows.

We have more than 50 Kerberos servers in 4 continents, with two servers for each major geographical area. All the servers support our one worldwide domain. We desired to have database consistency such that any change would be replicated worldwide within seconds. Hierarchical propagation allows data to flow to intermediate servers which then distribute data to other servers in the continent.

Morgan Stanley required similar incremental propagation software from the Kerberos 5 vendor. Our user transparency requirement dictated constant robust synchronization between the Kerberos 4 and Kerberos 5 databases. We built a gateway that incrementally propagates Kerberos 4 database changes to the Kerberos 5 propagation tree. The gateway does not propagate Kerberos 5 database changes back to the Kerberos 4 world. Therefore, during the transition period from Kerberos 4 to Kerberos 5, users are limited to the Kerberos 4 variants of database update utilities (kpasswd, kadmin), preventing changes to the Kerberos 5 databases. These utilities will be replaced by their Kerberos 5 equivalents at the completion of the deployment. Thus, during the transition, the Kerberos 4 master is the global master KDC.

4. Kerberized Tools

4.1 Kerberized Underuser Utility

Many UNIX professionals are familiar with sudo, a publicly available utility that delegates higher privileges to specific users. Prior to the introduction of Kerberos, Morgan Stanley implemented a conceptually similar program called 'uu' (underuser), which allowed administrators to grant limited superuser privileges to certain users. This was used, for example, to give support personnel the ability to change the passwords or kill the screen lock of only those users they supported, or to give developers the ability to run network snooping utilities on some machines.

With Kerberos 5, we extended 'uu' to allow privileged users access to the Kerberos and AFS credentials of other users. The Kerberized underuser program, 'kuu', is quite flexible: users can be restricted to acquiring the Kerberos credentials of others to only perform specific tasks. This is in contrast to ksu, the Kerberized su utility, which only supports complete access to a target user's account, or no access at all. Also, when kuu requires a password, the password entered is the user's own password, not the target user's password as in ksu. kuu privileges are controlled by a centrally maintained authorization file, which contains lines with the following information:

key:domains:hosts:underusers:subject:runas:ticketas:passreq:command with args

The key specifies the kuu function to be performed; each field thereafter further restricts the ability of an underuser to perform this function. domains are the NIS domains in which these underusers can perform this function. The hosts field lists the hosts on which these underusers can perform this function. underusers lists the users allowed to perform this function. subjects are the users upon which the function may act. runas is the user which the command specified in the following field will be run as. ticketas specifies the principal for which a ticket will be forwarded. passreq defines whether the underuser must reenter their password to run the command. command with args is the actual command to be executed.

A few examples demonstrate the flexibility of the configuration file:

```
etherfind:*:machine1,machine2:tannoc,hollande:*:root
:{}:*:/usr/etc/etherfind $*
```

The etherfind rule allows users tannoc and hollande to execute etherfind as root on hosts machine1 and machine2. The '{}' in the ticketas field simply indicates that no Kerberos tickets are passed back to the kuu client, as none are required for etherfind. The '$*' in the command field indicates that whatever command line arguments are specified when invoking kuu are passed on to etherfind.

```
afsdist:*:*:hollande:*:afsadmin:afsadmin:*:/usr/local/b
in/vms dist $*
```

The afsdist rule allows user hollande to distribute volumes in the AFS file system using vms, our internal Kerberized client-server volume management system, with superuser (system:administrators) AFS credentials.

```
newpass:*:*:tannoc:+netgroup1:root:{}:*:/usr/local/etc
/newpass {s}
```

The newpass rule allows user tannoc to change the password of any member of the netgroup1 netgroup. The '{s}' in the command field is replaced with the subject of the kuu command.

Kuu is a client-server program. The kuu client, invoked as 'kuu subject key args', uses the user's Kerberos 5 credentials to mutually authenticate to the kuud server. Thereafter, all communications between the kuu client and kuud server use KRB_PRIV encrypted messages. The kuu client sends its command line arguments, as well as the machine and NIS domain on which it was invoked, to the kuud server. The kuud server uses this information with the configuration file to verify the clients authorization to perform the requested action. Successful authorization is communicated back to the kuu client, which executes the requested action on behalf of the user. If the requested action requires the valid Kerberos credentials of another user, the kuud server extracts the secret key of that user from the Kerberos database, obtains a Kerberos ticket for that user, and forwards it to the kuu client. The kuu client uses the forwarded ticket to obtain AFS credentials.

4.2. Automatic Srvtab Installation

A driving force behind the design of Morgan Stanley's UNIX environment has been the need to reduce support costs and ease system administration. Most UNIX machines are completely configured via network boot. Should a financial trader's workstation fail due to suspected hardware problems, support staff can immediately swap the machine with a spare and execute a network boot command to quickly partition disks and load an operating system. Network booting also simplifies installation of new machines: the necessary configuration can be done centrally by senior system administration staff; local support personnel need only connect the computer's cables and execute a network boot command. Finally, system administrators use network booting to quickly rebuild large groups of machines during non-business hours with new or upgraded operating systems and configurations.

In this model, minimal manual intervention is required to rebuild a machine. However, secure installation of Kerberos 5 srvtab files becomes very difficult. Normally, administrators authenticate themselves on the local machine and execute certain kadmin commands to extract the srvtab. Srvtab files can theoretically be made available over the network; however, they would clearly be open to compromise.

Morgan Stanley's approach was to build a client server program, dst (download srvtab). dst contacts a daemon running on a Kerberos 5 server that constructs the appropriate srvtab and forwards it back to the client for installation. While the dst client cannot rigorously authenticate itself to in.dstd, and cannot therefore protect the srvtab file in transit over the network, the server does check that the client has bound to a reserved port, indicating that it is running with superuser privileges, and only provides the srvtab corresponding to the IP address of the client.

The automated srvtab installation issue is a subset of the larger issue of workstation authentication in distributed systems. In order to authenticate itself, a workstation must possess some secret by which it can prove its identity, or have an administrator vouch for the identity of the machine. Lampson et. al. describe a method by which a such a secret could be stored in nonvolatile memory [Lampson et. al. 1992]. Such a system would for a large scale implementation likely be quite expensive, hard to administer, and would still be unsatisfactory, as secrets would still need to be installed on new machines shipped from the hardware vendor. Having administrators authenticate themselves on the local machine to extract the srvtab is equivalent to having that administrator vouch for the identity of the machine, but this is also unacceptable in our environment. Since these alternatives are not possible, we have settled on dst as an adequate compromise between security and operational convenience.

4.3. Kerberized Cron Utilities

Many cron jobs at Morgan Stanley require access to Kerberos credentials in order to execute Kerberized utilities such as rsh/rlogin/rcp, or to access protected directories in the AFS filesystem. In our Kerberos 4 environment, such cron jobs execute kinit with a password stored in a file. Such password files are difficult to protect using standard operating system file security. Whenever employees with knowledge of these passwords leave the Firm, the passwords must be changed and the files updated, an operational headache. Finally, our vendor's Kerberos 5 kinit program does not accept non-tty input (a desirable security feature), making this password-store-in-file approach infeasible as well as unwise.

To avoid these difficulties, Morgan Stanley implemented a more secure, scaleable solution for Kerberized cron jobs. Cron jobs use Kerberos 5 tickets stored in a special location (/var/spool/tickets) on the local workstation that have short actual lifetime (typically one week) but are renewable for up to a year.

A generic wrapper for cron jobs, kcron, obtains Kerberos 4 credentials and AFS tokens via krb524, sets the KRB5CCNAME environment variable, and then executes the cron job. Because the tickets in /var/spool/tickets are also forwardable, cron jobs that also execute commands on remote machines (via rsh, for example) can forward them as necessary.

Consider the following crontab entry for user sysmon, which runs a Kerberized program to monitor system health.

```
# Execute the sysmon script at 4:00 AM with Kerberos credentials.
0 4 * * * /usr/local/bin/kcron /usr/local/scripts/sysmon host > /dev/null 2>&1
```

4.3.1 Renewing Tickets

Because tickets stored in /var/spool/tickets have a short actual lifetime, they must be renewed on a regular basis to prevent expiration. We accomplish this by having a simple script that uses kinit to renew all tickets in /var/spool/tickets. This script runs, as root, weekly on every UNIX machine at Morgan Stanley as part of our overall system maintenance processes. This ensures that Kerberos tickets used by kcron processes remain valid from week to week and reduces the risk of critical batch streams failing due to expired tickets.

4.3.2 Ticket Installation

As described in section 5.1, most UNIX machines at Morgan Stanley can be completely configured via network boot. This presents a problem for kcron tickets, as newly scrubbed machines will not have the necessary tickets installed in /var/spool/tickets, causing kcron jobs to fail. To solve this problem, we have implemented a client-server program, run as part of the network boot procedure, that installs the necessary kcron tickets in /var/spool/tickets. This program is also used to install new tickets on a machine; since the Kerberos principals corresponding to production ids have a random secret key not derived from any password, kinit cannot be used to install tickets.

Ticket installation is performed by the krbtk client, which contacts the in.krbtkd server running on Kerberos 5 database servers. The in.krbtkd server consults a central registry to determine which tickets should be installed on the client machine. It extracts the secret keys for all these principals from the Kerberos 5 database, and returns them to the client. The client then obtains Kerberos tickets for all principals and stores them in /var/spool/tickets. The krbtk client, which must run as root, authenticates to the in.krbtkd server using the local machine's host principal; all messages passed between client and server are encrypted in KRB_PRIV messages.

4.3.3 Design Alternatives

We faced a few design alternatives when building kcron. One alternative was simply to have cron jobs that require Kerberos credentials authenticate via a key stored in a keytab file. While this approach is appealing due to its simplicity (no infrastructure for ticket renewal is necessary), it is little better than passwords stored in files - the srvtabs must sill be updated when the secret key for a principal is updated.

Another alternative was to simply issue tickets with long actual lifetimes (a year). This would also obviate the need for weekly ticket renewal. This solution forgoes the security advantage of renewable tickets, namely that, should a ticket be reported stolen, the KDC may refuse to renew tickets, thereby thwarting their continued use [Neuman & Kohl 1993]. However, this is of doubtful practical value. In an environment of 4000 UNIX machines, with frequent and ubiquitous use of Kerberos, simply detecting a single stolen Kerberos ticket is a daunting challenge. In addition, our vendor-supplied KDC does not allow administrators to deactivate ticket renewal for a particular principal (short of deleting the principal in question from the Kerberos database), nor, to our knowledge, does the MIT KDC.

4.3.4 A Comparison to lat

Rubin and Honeyman [Rubin & Honeyman 1993] describe a system for long running jobs in an authenticated environment. While lat and kcron are quite different in implementation, they operate on the same principle: some type of credential is left around on the local machine that authenticates the job. In the case of lat, it is a session key encrypted with a random key; in kcron, it is a renewable ticket. While a ticket is a more dangerous credential to leave on an unattended machine, this danger is mitigated, at least in theory, by the limited actual lifetime of the tickets. The lat client and latd server communicate to renew tickets if the batch job is to run for more than the lifetime of the ticket granting ticket (TGT); with Kerberos 5 this feature is inherent and kcron makes use of it directly. Finally, while lat seems applicable only to at-style jobs,

kcron can be used for both cron jobs and at jobs, although using postdated tickets may be more appropriate for at jobs in a Kerberos 5 environment. The use of postdated tickets for at has not been investigated because of the limited use of at jobs at Morgan Stanley.

5. Kerberized Application Tools

Kerberizing infrastructure tools allow application developers to build secure applications. We developed and internally published a security API that provided functionality typically needed by our developers. We considered using GSSAPI [Linn 1993], but rejected it because it was complicated and it did not support ticket forwarding at that time.

We have a Kerberized RPC mechanism that facilitates data transfer between mainframe databases and client applications on UNIX workstations. Numerous critical applications use this facility to securely access the mainframe. This facility currently uses Kerberos 4, but we plan to migrate this code to use Kerberos 5 in the near future.

Our internal broadcast data distribution system is Kerberized. This allows us to limit access to market data as well as internal business data to authorized users. Applications built on top of this distribution system inherit the security built into it.

Kerberos also allows us to build a single signon solution for our Sybase servers. We have a Kerberized Sybase "password server" that stores Sybase passwords for users that are authorized to use a database. The Sybase client program securely retrieves this password using Kerberos based authentication and logs into Sybase before transferring control to the regular SQL program. Unfortunately, this mechanism is cumbersome and not useful for all applications.

6. Future Plans

We intend to extend our single sign-on capability to our mainframes as well using a feature known as 'passtickets'. During user authentication, the mainframe security system can be configured to accept a one time passticket instead of the regular user password. Such passtickets can be generated by a passticket server program running on a UNIX host. The mainframe terminal emulator program will be modified to use Kerberos to securely obtain a passticket, and then send it to the mainframe. The user will be automatically logged in by virtue of having a Kerberos ticket.

One of the criteria for selecting our vendor was the availability of Kerberos software on PCs. This should allow us to Kerberize our client-server applications that run on PCs. Kerberizing some PC desktop machines require a different strategy than was used for UNIX desktops. Many of our PCs are docking stations or laptops that often run disconnected from the network. It is not practical to Kerberize the screenlock on these PCs, because the KDC may not be online to verify the password.

Another goal is to make Kerberized HTTP available within the firm for use by internal applications. This would allow us to control access to sensitive data on our internal web servers using the existing Kerberos authentication infrastructure. Much of this work has been done but unfortunately is not available on the web browsers and servers that we use.

7. Conclusions

For Kerberos to succeed in mission-critical production environments, vendors need to think on a large scale. Tools appropriate for smaller environments are not always appropriate in environments where failure of a batch process could cost millions of dollars. Automation is crucial: we created tools to automate srvtab installation and to monitor the health of all Kerberos processes. Special attention was paid to incremental propagation stability. We also developed our own Kerberized cron mechanism. We integrated Kerberos seamlessly into the environment to minimize any user inconvenience. Finally, we changed the default maximum credential lifetime in the Kerberos database, as the default value was too short to be useful. These are some examples of enhancements we felt were essential in integrating vendor Kerberos products into Morgan Stanley's distributed computing infrastructure. We learned to make effective compromises between security, usability and administrability.

Crypto-based authentication systems should be well integrated into all aspects of modern operating systems, messaging systems, and distributed applications. Operating systems should ship with hooks in basic authentication programs and easy-to-use libraries for application use. Operating system vendors should design a general, highly configurable scheme to provide for alternate authentication mechanisms, such

as Kerberos, SecureID, S/Key, etc. Our solution of defining '*' to mean a Kerberized user is not general enough. The Pluggable Authentication Module specification from Sun appears to be gaining industry acceptance; however, more work remains to be done in developing general and configurable authentication subsystems. The same applies to application vendors who supply mail agents, databases, groupware, and other applications. All should be provided with hooks or exits at critical points for customer provided authentication or authorization checks.

We spent approximately 2 person years to adequately integrate the vendor provided software into our environment. Many companies may be reluctant to devote this much effort to solve the authentication problem in a world of rapidly changing technology. However, Morgan Stanley's strong commitment to open distributed systems, in balance with proper information security controls, justified this effort.

8. Acknowledgments

We thank James Anderson, David Bauer, Benjamin Fried, Douglas Kingston, and W. Phillip Moore for their helpful comments and suggestions during the course of this work.

9. References

1. M. Bellovin and M. Merritt, Limitations of the Kerberos Authentication System, in *USENIX Conference Proceedings*, pages 253-267 (Dallas, TX; Winter 1991).
2. D. Engert, Aklog and Kerberos 5, *message to the info-afs@transarc.com mailinglist*, 11/7/94
3. D. Engert, Aklog for Kerberos 5, *message to the info-afs@transarc.com mailinglist*, 1/17/95
4. X. Gittler, W. P. Moore, and J. Rambhasker, Morgan Stanley's Aurora System: Designing a Next Generation Global Production UNIX Environment, in *Proceedings of the Ninth System Administration Conference (LISA '95)* (September, 1995).
5. B. Lampson, M. Abadi, M. Burrows, and E. Wobber, Authentication in Distributed Systems: Theory and Practice, *ACM Transactions on Computer Systems* 10(4) (November, 1992)
6. J. Linn, *Generic Security Service Application Program Interface*, Internet RFC 1508, September, 1993
7. C. Neuman and J. Kohl, *The Kerberos Network Authentication Service (V5)*, Internet RFC 1510, September 1993
8. A. D. Rubin and P. Honeyman, Long Running Jobs in an Authenticated Environment, *Proc. 4th USENIX UNIX Security Symposium*, Santa Clara (October 1993)

A Framework for Building an Electronic Currency System

Lei Tang *

GSIA
Carnegie Mellon University
Pittsburgh, PA15213-3891
ltang@cmu.edu

Abstract

We describe the framework for building an electronic currency system. We detail the design of the components of the electronic currency system and the relationship among them. Contrary to the previous electronic currency literatures, which focus exclusively on electronic currency protocol designs, we address how to achieve both transaction atomicity and transaction anonymity at the presence of the hostile failures, which are common in an electronic currency system if the customers or the merchants are dishonest or malicious. We also propose a recovery method called redo_transaction to recover from hostile failures so that the aborted electronic currency transactions caused by the hostile failures can be forced to commit eventually. The structure of the electronic currency system is protocol-independent in the sense that those Chaum-like off-line electronic currency protocol could be incorporated into our framework.

1 Introduction

An electronic currency system is the strongest electronic payment system in the sense that it provides anonymity (hence privacy) to its customers, as its counterpart paper currency does. One difference between the electronic currency and the paper currency is that the payer and the payee who complete a transaction through the electronic currency do not have to be face-to-face, as the case in a paper-currency transaction. The big challenge for building an electronic currency system is to build a system secure against malicious attacks from the adversaries (including both attacks on the system and cryptoanalysis attacks on the protocols) while preserving the transaction atomicity, which guarantees the consistent and non-repudiational termination of the transaction. Electronic currency protocol design and transaction processing management are two key components of an electronic currency system. Most of the work so far in the electronic currency area are concentrated on the protocol design, while little has been reported on the overall structure of an electronic currency system, and the foundation for building such a system–transaction management.

There are two implemented electronic currency systems: the DigiCash system by Chaum and the NetCash system [13] by Neuman and Medvinsky. The former is based on Chaum's blind signature technique [4]. The latter is based on a trusted third-party. It is an on-line system based on private key algorithms. Due to the protection of commercial secrecy, little has been reported on the structures of the systems and transaction management mechanisms. In this paper, we present a framework to build an electronic currency system. Our framework is protocol-independent so that the existing Chaum-like off-line electronic currency protocols could be incorporated into this framework.

There are many problems faced by electronic currency system designers and builders. We list a few that we think are crucial for building an electronic currency system.

- Electronic currency protocol design and protocol security analysis.
- Public key management, especially real-time key revocation since the electronic currency system is off-line.
- Secure and verifiable message transmission through an open computer network.
- Transaction management, especially how to

*The views and conclusions contained in this document are solely those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Carnegie Mellon University.

achieve both atomicity and anonymity; how to recover from hostile failures caused deliberately by malicious participants involved in the transaction. Anonymity requirement of an electronic currency system makes building a robust recovery mechanism difficult.

It is impossible for us to address all these issues in a single paper. Instead, we focus on the description of the overall structure of the off-line electronic currency system, the transaction management, and the recovery mechanisms (especially recovery from hostile failures).

A concept called verifiable transaction atomicity [15] is used to design our system so that both anonymity and transaction atomicity can be achieved. We also use a technique called redo_transaction to recover from those hostile failures so that the aborted transactions caused by the hostile failures will be forced to commit eventually.

Our presentation is organized as follows. We give an overview of the structure of our system in section two. In section three, we give brief introduction to the electronic currency concept, and the verifiable transaction atomicity concept used to achieve both transaction atomicity and transaction anonymity. We detail the structure and the components of the bank service, and the relationships among all these components. Section five and section six describe the customer services and merchant services respectively. We conclude in section seven.

2 Overall structure of the system

Our system consists of parties belonging to three different categories. They are a bank, a group of customers, and a group of merchants. The bank issues electronic currency tokens to the customers and receives electronic currency tokens from the merchants for deposit. Once the electronic currency token is issued to the customer by the bank, the identity of the customer associated with this token is untraceable if the customer does not use the token twice. If the customer reuse the electronic currency token, his identity will be revealed by the bank when the merchant deposits this token to the bank. The customers and the merchants complete their financial transactions by passing the electronic currency tokens from one party to another. The customer's identity should not be revealed to the merchant during the payment phase. The merchants deposit those electronic currency tokens obtained from the customers to the bank for refund in the deposit phase.

For our purpose, each customer or merchant has a computer running an operating system that is trusted for local security and is not shared with other users. The bank provides banking services to the customers and the merchants. The banking services are well protected. All of the nodes in the distributed system are connected by an insecure network. Different nodes communicate by sending and receiving messages over the links of the network. We assume the system is synchronous in the sense that bounds exist (and are known) for both relative speeds of processes and message delays, so the communication failures can be detected by a timeout mechanism (the detailed definition of synchronous is given in [2, 10]).

At any time a process is either faulty or correct. A process is faulty in an execution if its behavior deviates from that prescribed by the algorithm it is running. Otherwise, it is correct. A model of failure specifies in what way a faulty process can deviate from its algorithm. There are several types of process failures. They fall into two broad categories: the benign failures and the hostile failures. The definition of a *benign failure* is given in [10]. A process that suffers a benign failure does not arbitrarily change state, or send an arbitrary message that is not prescribed by its algorithm according to its present state. We call failures with more severe faulty behaviors than those faulty behaviors exhibited by benign failures the *hostile failures*. The benign failures (such as system crashes) are the ones that arises most commonly where no monetary transactions are involved. Most of the fault-tolerant protocols in distributed systems are focused almost exclusively on those benign failures. In the electronic currency system, a dishonest or malicious process may exhibit any behaviors which will help itself to benefit financially from its behaviors by sabotaging the transactions. The failures (caused by a faulty process) exhibiting an arbitrary behavior are hostile failures (also called Byzantine failures, or arbitrary failures). With arbitrary failures, a faulty process may claim to have received a particular message from a correct process, even though it never did; a faulty process may claim to have never received a particular message from a correct process, even though it did; or a faulty process may fabricate failures appeared to be benign failures any time during the transaction.

The setting for our system is a distributed system where each node runs the Unix operating system. Some distributed transaction facility such as the Encina family of products developed by Transarc Corporation is layered above the operating system. It provides mechanisms for constructing reliable distributed programs that are robust against benign

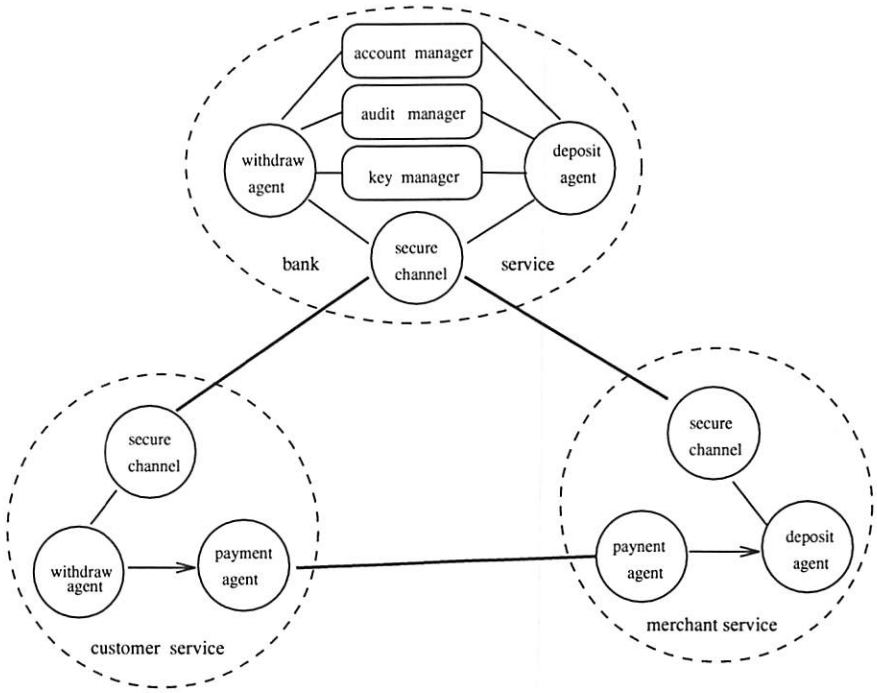


Figure 1: The structure of the electronic currency system

failures and guarantee transaction consistency.

In each node, a new operating system component called the *secure communication agent* manages identity authentication of the party with which the node communicates, and establishes a secure channel between them. All subsequent communication associated with this session has to go through this secure channel. There are two different currency withdrawal agents located in the bank's node and the customer's node respectively for the customer to withdraw electronic currency tokens from the bank. There are also two different currency deposit agents located in the bank's node and the merchant's node respectively for the merchant to deposit electronic currency tokens to the bank. The customers and the merchants do transaction through the payment agents in them. The bank also has other management servers for key management, account management, and audit management. The structure of our system is shown in Fig 1.

3 Background

In this section we give a brief introduction to the electronic currency concepts, the protocol designs, and the special properties. Then we explain the difference between the atomic commitment problem in an electronic currency system and that in the traditional database system. We describe a concept called *verifiable transaction atomicity* [15] and briefly describe its specifications. We also define some notations used throughout this paper.

3.1 Electronic currency concept and protocol

An electronic currency token denotes money by a sequence of bytes. An ideal electronic currency system should have properties that the paper currency has: unforgeable, uncopyable, untraceable, off-line between the merchant and the bank (so that the merchant does not have to consult the bank to check if an electronic currency token is valid or not whenever the merchant receives a token from a customer), and transferrable from one customer to another. Uncopyability, unforgeability, and untraceability properties make the design of an electronic currency system challenging. Paper currency has physical properties that add security (it is difficult to copy or counterfeit dollar bills). Electronic currency is only represented as a sequences of bytes. It is harder to prevent copying, counterfeiting, and tracking the flow of electronic currency.

Chaum introduced the *blind signature technique* [4] to solve the problem of untraceability. The basic idea of Chaum's blind signature is as follows: Let (r, n) and (d, n) be the *RSA* public key/secret key pair for a bank. When a customer wants an

electronic token worth one unit, he chooses a random number x and a number y of a special form. Then he sends $C = f(y) \cdot x^r \bmod n$ to the bank, here f is a publicly known one-way hash function. After receiving C, the bank sends $D = C^d \bmod n$ to the customer. The customer computes the value of $f(y)^d \bmod n$ from $D = f(y)^d \cdot x \bmod n$ since the customer knows the value of x. When some merchant presents $(y, f(y)^d \bmod n)$ (sometimes denoted by $(y, f(y)^{1/r} \bmod n)$) to the bank, the bank can not determine which customer spent this token. Here the untraceability is achieved through the blind factor x generated randomly by the customer. We use the terminology *blind manipulation* to describe the technique of hiding values by multiplying them by random factors.

Chaum's original system did not protect against a customer making multiple copies of an electronic token and using it with multiple merchants. It is possible to solve this problem by requiring that the bank be on-line and participate in every transaction, but this raises difficult scaling and reliability issues. Using the cut-and-choose technique first introduced by Rabin [14], Chaum, Fiat, and Naor improved Chaum's original protocol by using an interactive protocol to determine if an electronic token was reused [5]. In the Chaum-Fiat-Naor protocol, many special values are computed by a customer and sent to the bank. The bank issues a cryptographic challenge, and the customer replies after performing additional computations. A set of lightweight electronic currency protocols are designed after the Chaum-Fiat-Naor protocol [6, 7, 13]. We call all those off-line electronic currency protocols which are based on RSA algorithm and Chaum blind signature the Chaum-like protocols. In this paper, we design an electronic currency framework which could accommodate all these Chaum-like protocols. In the sequel, we use $f(y)^{1/r}$ to denote the electronic currency token in the Chaum-like electronic currency protocols for simple exposition.

3.2 Transaction Atomicity and Anonymity

An electronic currency system should have anonymity (or untraceability) properties. Basically, the electronic currency token should be anonymous, which means that the token can not be associated with the identity of the customer once it is issued by the bank. Also the identity of the customer should be anonymous when the electronic currency token is paid to the merchant. The anonymity requirement makes transaction atomicity difficult to accomplish. For example, in the currency withdrawal protocol, a customer sends a currency withdrawal request to the bank. The bank authenticates the customer's identity. If the authentication is successful, the bank issues an electronic currency token to the customer and deducts the customer's account. In this protocol, a dishonest customer may cheat as follows. The dishonest customer does not sign an acknowledgement to the bank server to commit the transaction after he receives the currency token. Instead, the customer fabricates a failure (e.g., he unplugs the network connection to pretend a communication failure) deliberately so that there is no way for the bank server to detect the cause of the failure. Hence the bank server can not determine if the electronic currency token has been received by the customer or lost in the transmission. If the transaction is managed by a traditional transaction manager (according to the two-phase commitment protocol [2, 8, 11]), the bank will abort the transaction and undo the updates on this customer's account. The dishonest customer obtains a valid currency token, but his bank account is not deducted. Since the electronic currency token is untraceable, the identity of the dishonest customer can not be revealed if he spends the electronic currency token once. The bank will lose money because of the dishonest behavior of the customer during the withdrawal procedure. This is unacceptable to the bank. This is also an example that the traditional transaction management techniques used to achieve transaction atomicity are not robust enough to handle hostile failures caused by malicious or dishonest participants involved. To remedy this problem, a concept called *verifiable transaction atomicity* is proposed in [15]. Beside the properties which the atomic commitment protocol should have [9, 1], the verifiable atomic commitment protocol should have two additional properties.

- The interactions (communications) and the contents of the interactions among all participants are verifiable (by a third party) if necessary.
- If a participant receives a message with which some monetary value is associated, the transaction in which the participant is involved will be committed eventually (even though the participant may abort the transaction unilaterally).

Verifiable transaction atomicity guarantees both transaction atomicity and transaction anonymity.

3.3 Notations

We use the symbols S and R to denote the sender and the receiver respectively. T_a represents the times-

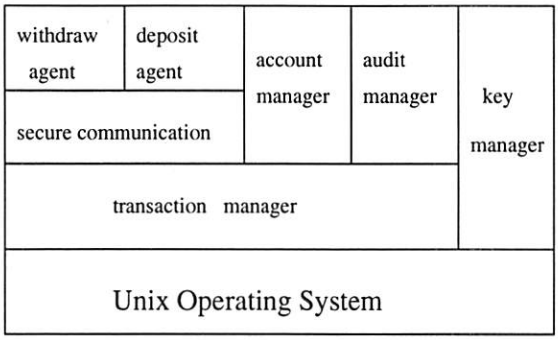


Figure 2: The structure of the bank service

tamp read from A's clock. We also use the symbols below to represent the following.

K_a A's public key.
K_a^{-1} A's private key.
K_{ab} The key shared between A and B.
$\{x\}_K$ x encrypted with key K.
x, y x concatenated with y.
Id_a A's true identity.
P_a A's pseudonym.

4 The bank service

Our electronic currency system consists of three parts: the bank, the customer, and the merchant. The bank provides services for authentication and secure communication with the customers and the merchants. It also provides services for currency withdrawals with the customers and currency deposits with the merchants. The customers (the merchants) withdraw (deposit) currency with the bank service via the client process running in the customers' (the merchants') local node. The bank services consists of many agents implementing all token exchanges and validations. All of them can be implemented as independent servers. They are the *secure communication agent*, the *withdrawal agent*, the *deposit agent*, the *key manager*, the *account manager*, the *audit manager*, and the *transaction manager*. The structure of the bank service is shown in Figure 2.

The secure communication agent is responsible for authenticating the customer's (or the merchant's) identity and establishing a secure channel between the bank and the customer (or the merchant). All subsequent communication has to go through the secure channel. It is also responsible for achieving verifiable message transmission, which will be elaborated later in this section. The withdrawal agent generates electronic currency tokens securely for the customers. The deposit agent receives and verifies tokens deposited by the merchants. The account manager is responsible for updating bank accounts for the customers and the merchants. The transaction manager provides fault-tolerance and consistency to our system. The audit manager audits all system activities and logs these operations to the hard disk efficiently. The key manager is responsible for registering and revoking public keys for the customers and the merchants. We will describe the structures of the secure communication agent, the withdrawal agent, the deposit agent, and the transaction manager in the following sections.

4.1 The secure communication agent

The secure communication agent handles the following functions.

- Authenticating the identity of the customer (or the merchant) running the client process.
- Generating a secure session key known only to the bank service and the client process for currency withdrawal transactions and currency deposit transactions. This session key is cached during the transaction and is flushed after the transaction. We use **secure-send**(m) and **secure-receive**(m) to denote the secure communication primitives.

Two verifiable message transmission primitives **delivery**(m, *receiver*) and **receipt**(m, *sender*) are designed to achieve two properties. First, if the receiver receives a message m from the sender, it can not deny the receipt of m, or the content of m. Second, if the sender transmits a message m to the receiver and the receiver receives it, the sender can not deny the sending of m or the content of m. The two primitives corresponding to that the sender transmits a message m to the receiver and the receiver receives a message m from the sender respectively. The secure communication agent is responsible for verifiable message transmission. We use the pseudo-Pascal syntax with the usual sequential control flow structure to describe these two primitives.

The **wait-for** statement is used to block the message receiver until the receipt of a particular message. If the message may arrive at unspecified times and should be received without blocking the message receiver, then the **upon** statement is used. δ is the upper bound on the message delay, this consists of the time it takes for sending, transporting, and receiving a message over a link. In case of a blocking wait, an optimal timeout may be set to trigger at

a particular (local) time using the **set-timeout-to** statement. The timeout value in effect is that set by the most recent **set-timeout-to** before the execution of a **wait-for** statement. If the event being waiting for does not occur by the specified time, then the **on-timeout** clause of the **wait-for** statement is executed rather than its body. The body and the time-out clause of wait-for are mutually exclusive.

We assume that there are some daemon processes (e.g., a httpd server, a ftpd server, or a gopher server) in the sender's side to export a portion of the local file system. The **location** sent to receiver by the sender could be a Universe Resource Locator (URL). In case that the sender wants to prevent the participants other than the receiver from accessing the information located at **location**, the file name of the URL can be randomized so that only the intended receiver can obtain the URL. The content of the location exported by the sender can be verified by a third party if the receiver disputes the content of the file in **location**.

In the algorithm, a certificate to the public key is sent together with any message signed by the corresponding private key. We do not write down explicitly the certificate for simple exposition. In case that the anonymous sender nonrepudiation property is desired, the pseudonym [3] of the sender P_s is included in the message and certificate instead of the true identity of the sender. The protocol of verifiable message transmission is shown in Figure 3 [1].

The protocol consists of two tasks, one is executed by the sender (the procedure **delivery**) and the other executed by the receiver (the procedure **receipt**). The sender starts by sending out the message with an encrypted form and waiting for the signed acknowledgement of the receipt of the encrypted message from the receiver. When the receiver receives such a message, the receiver sends out the signed acknowledgement. After receiving the signed acknowledgement from the receiver, the sender sends the decryption key to the receiver and puts the decryption key on the URL **location** no matter whether the sender receives an acknowledgement about the receipt of the key from the receiver or not. If the sender does not receive the signed acknowledgement in time, the sender will unilaterally decide to abort the delivery. After receiving a message, the receiver checks if the message has the specific form signed by the sender, then the receiver responds with a signed acknowledgement and waits for the arrival of the decryption key. If the receivers receives the key from the sender directly, he can obtain the message m.

[1] Figures 3-6 are all borrowed from [15]

SENDER:
procedure delivery(m, receiver)
  invent K;
  $Em := \{\{m\}_K, Id_s, Id_r, T_s, \textbf{location}\}_{K_s^{-1}}$
  secure-send(Em) **to** receiver
  set-timeout-to $T_s + 2\delta$
  wait-for (**secure-receive**(Dm) **from** receiver)
    secure-send ($\{K, T_s, Id_r\}_{K_s^{-1}}$) **to** receiver
    $Ek = \{K, Id_r, Id_s, T_s\}_{K_s^{-1}}$
    put Ek into **location**
    log(signed receipt of m from receiver)
  on-timeout
    abort delivery
    decide according to terminate-protocol()

RECEIVER:
procedure receipt(m, sender)
  upon (**secure-receive**(Em) **from** sender)
    verify if Em is in the proper format
    $Dm := \{\{m\}_K, Id_r, Id_s, T_r\}_{K_r^{-1}}$
    secure-send(Dm) **to** sender
    set-timeout $T_r + 2\delta$
    wait-for (**secure-receive**(Ek) **from** sender)
      $Dk := \{K, T_r, Id_r\}_{K_r^{-1}}$
      secure-send(Dk) **to** sender
      decrypt $\{m\}_K$
    on-timeout
      fetch K from **location**
      decrypt $\{m\}_K$

Figure 3: Verifiable Message Transmission Protocol

Otherwise, he fetches the key located in **location**, which is sent to him by the sender in the first round of the communication. The sender may send a false key K' to the receiver, or put a false key K' in **location**. But he gains nothing since the sender has to give k to the receiver under a trusted arbitrator.

Besides the procedures **delivery** and **receipt**, we also provide a procedure **anon_delivery**. It is the same as **delivery** except that the sender's pseudonym P_s instead of his true identity Id_s is included in the messages and the public key certificate passed between the sender and the receiver. **anon_delivery** achieves the anonymous sender nonrepudiation property.

The secure communication agents in the customer service and the merchant service also implement the above verifiable message transmission primitives.

4.2 The withdrawal agent

The currency withdrawal agent processes the currency withdrawal operations between the customer and the bank. After the customer's identity is au-

thenticated, the withdrawal agent issues electronic currency tokens to the customer and informs the account manager to deduct the customer's account.

All withdrawal operations inside the withdrawal agent are bracketed by **begin_transaction** and **end_transaction**. The withdrawal agent has the functions described as follows.

- Executing a currency withdrawal protocol with the customer and issuing the currency tokens to the customer.
- Notifying the account manager to deduct the customer's account after the currency tokens are issued or to credit the merchant's account after the currency tokens are deposited.
- Inquiring the key manager about the validity of the customer's public key.

The currency withdrawal agent implements a secure currency withdrawal protocol between the customer's client (called C) and the bank service (called B). After the withdrawal protocol is completed, the customer obtains a fresh electronic currency token.

The **auth** and **verify** modules are procedures of authentication between the sender and the receiver. **redo_transaction** is the procedure to force an aborted transaction to commit. We describe **redo_transaction** in section 4.4. **atomic_commitment** is a generic atomic commitment protocol [1, 2, 8] with two additional operations when the transaction is aborted: the identities of the participants are logged into the redo_list and the redo records are copied to the redo_log when the transaction is aborted. A YES vote indicates that the local execution was successful and the participant has received the nonrepudiational receipt, and that the participant is willing and able to make the updates to the data permanent. In other words, the updates and the nonrepudiational receipt have been written to the stable storage so that they can be installed as the new data values even if there are future failures. A NO vote indicates that for some reason (e.g., storage failure, deadlock, invalid monetary token, etc.) the participant is unable to install the results of the transaction as the new permanent data value. In the withdrawal protocol, the redo_transaction forces a dishonest customer to commit a transaction if he did receive a valid electronic currency token, which is untraceable once issued; or to complete a transaction if the number $xf(y)^{1/r}$ in which the electronic currency token is included gets lost during the transmission due to some network malfunctions. The withdrawal agents in the bank service and the customer service are shown in Figure 4.

withdrawal agent in the bank service:
procedure withdraw(Id_c, Id_b)
  upon receipt of withdrawal request from Id_c
  If (**verify**(Id_c)= success) **Then**
    If ($Id_c \in redo\_list$) **Then**
      redo_transaction(Id_c)
    Else
      receipt($x^r f(y), Id_c$)
      compute $xf(y)^{1/r}$
      delivery($xf(y)^{1/r}, Id_c$)
    set-timeout-to $T_b + 2\delta$
    wait-for(the receipt of Dk)
      update Id_c's account
      vote YES
      atomic_commitment($trans\_id, Id_c, Id_b$)
    on-timeout
      log Id_c to redo_list
      copy transaction log to redo_log
      vote NO
      abort($trans\_id, Id_c, Id_b$)

withdrawal agent in the customer service:
Procedure withdraw(Id_c, Id_b)
  sends withdrawal request to bank
  auth(Id_c)
  If (bank requests redo_transaction) **Then**
    redo_transaction(Id_b)
  Else
    receipt($xf(y)^{1/r}, Id_b$)
    If $f(y)^{1/r}$ is valid **Then**
      update account record and vote YES
      atomic_commitment($trans\_id, Id_c, Id_b$)
    Else
      vote NO
      copy transaction log to redo_log
      abort($trans\_id, Id_c, Id_b$)

Figure 4: Electronic Currency Withdrawal Agents

4.3 The deposit agent

The currency deposit agent processes the currency deposit between the merchant and the bank. The major part of the currency deposit agent is the replay checker. All deposited currency tokens are stored in the old token database. As time goes on, the database will grow bigger and bigger, and it takes longer to check if a token is a replayed one or not. One way to alleviate this problem is for the bank to change its public key at regular intervals. If a token is not used, the customer can ask the bank to exchange the old currency token for a new one. We delay the discussion of the deposit agent in section 6.

4.4 The transaction manager

The bank services consists of a group of servers to process the currency withdrawal operations between the customer and the bank, and to process the currency deposit operations between the merchant and the bank. This group of servers have to coordinate with each other and to be robust against failures, which occur frequently in the distributed environment. These failures include system crashes, network delays and partitions, failures caused deliberately by a malicious participant, and other unexpected failures. We have given such an example in section 2.3.

Transaction technique[2, 8] are usually used to build systems robust against those benign failures. Transactions were developed to solve problems caused by failures and concurrency. Failure recovery is achieved by a so-called two-phase commitment protocol and consistency is guaranteed by a so-called two-phase lock protocol. In our design, transaction is use mainly for failure recovery, so we will not discuss the consistency problem in our paper. Transactions have what are sometimes called the ACID properties: they are *Atomic*, *Consistent*, *Isolated*, and *Durable*. However, as we mentioned in Section 3.2, the traditional transaction management is not robust against those hostile failures.

We proposed a new method called *redo_transaction* to recover the failures (especially the hostile failures) and to force the participant who has received the electronic currency token to commit the transaction.

We give a sketchy description of how the redo_transaction procedure works. The redo_transaction forces two logs to the stable storage: the *redo_list* and the *redo_log*. The redo_list records the receiver's identity if the sender receives the signed acknowledgement of the encrypted electronic currency tokens but the transaction is aborted. The redo_log records the uncommitted transaction in which the sender has received the signed acknowledgement of the encrypted electronic currency token but the transaction is aborted. The redo_log is indexed by the unique identities of the receivers instead of the LSNs (log sequence numbers) in a usual transaction system. The redo_log is used to recover those failed transactions. The redo recovery in an electronic currency system is different from those redo algorithms focusing exclusively on recovering from a system crash [12, 2, 8]. The recovery mechanisms in a redo-transaction is more complicated than that in an ordinary transaction in the sense that the recovery mechanisms should not only handle recovery from transaction abort and system crashes, but also be able to process uncommitted transaction to force an aborted transaction in which the sender has received the signed acknowledgement of the receipt of the encrypted monetary token from the receiver to commit eventually. Redo_transaction guarantees that both anonymity and atomicity can be achieved in an electronic payment transaction even in the presence of malicious failures.

A redo_transaction begins with mutual authentication between the sender and the receiver. If the sender's identity is found in the redo_list, the logs of corresponding uncommitted transaction are fetched from the redo_log and are sent to the sender. The sender has to commit this transaction before the two parties can begin another new transaction. If the transaction is aborted, the transaction has to be redone until the transaction is committed.

payment agent in the customer service:
procedure payment(P_c, Id_m)
  anon_delivery($f(y)^{1/r}, Id_m$)
  set-timeout $T_c + 2\delta$
  wait-for(receipt of Dk and goods)
    update record and vote YES
    atom_commitment($trans\_id, P_c, Id_m$)
  on-timeout
    disputation_resolve()

payment agent in the merchant service:
procedure payment(P_c, Id_m)
  upon receipt of $f(y)^{1/r}, P_c$
    If $f(y)^{1/r}$ is valid **Then**
      delivery(goods)
      vote YES
      atomic_commitment($trans\_id, P_c, Id_m$)
    Else
      vote NO
      abort($trans\_id, P_c, Id_m$)

Figure 5: Electronic Currency Payment Agents

5 The customer service

The customer service consists of a secure communication agent, a withdrawal agent, and a payment agent. The secure communication agent authenticates the customer's identity to the bank and implements two primitives we described in section 4.1. The withdrawal agent obtains electronic currency tokens from the bank through the procedure shown in Figure 3. The payment agent pays the currency tokens to the merchant and obtains goods or services in return. We describe the payment agent in details in this section.

An important task of the payment agents in the

```
deposit agent in the merchant service:
procedure deposit(Id_m, Id_b)
    auth(Id_m)
    delivery(f(y)^{1/r}, Id_m )
    set-timeout-to  T_m + 2δ
    wait-for(the receipt of Dk)
      update account record
      vote YES
      atomic_commitment(trans_id, Id_m, Id_b)
    on-timeout
      disputation_resolve()

deposit agent in the bank service:
Procedure deposit(Id_m, Id_b)
    If (verify(Id_n)= success) Then
      receipt(f(y)^{1/r}, Id_m)
      If (f(y)^{1/r} is valid and not reused) Then
        update Id_m's account
        vote YES
        atomic_commitment(trans_id,Id_m, Id_b)
      Else
        vote NO
        abort(trans_id, Id_m, Id_b)
```

Figure 6: Electronic Currency Deposit Agents

customer service and the merchant service is to find who has reused an electronic currency token, the customer or the merchant, if the bank finds that an electronic currency token was reused. Two methods could be used to prevent a dishonest merchant from framing its customers, or to prevent a dishonest customer from reusing a token and denying it. One is to ask the merchant to check replay before it accepts an electronic currency token. Another method is to sign the electronic currency token during the payment. In the second case, the customer's identity should not be revealed to merchant while the merchant can validate the signature. The primitive **anon_delivery** is used to satisfy the above requirements. The primitive serves two functions. One function is to prevent the customer from denying reusing the electronic currency token if he did. Another function is to prevent a dishonest merchant from reusing an electronic currency token and framing that a customer had reused it. If an electronic currency token is detected to be reused by the bank (the details are shown in the deposit protocol), the bank can decide who reused it, the merchant or the customer, by asking the merchant to turn in the corresponding anonymous signature on the electronic currency token. The operations of the payment agents in the customer service and the merchant service are shown in Figure 5.

6 The merchant service

The merchant service consists of a payment agent, a secure communication agent and a deposit agent. I have discussed the payment agent and the secure communication agent in the previous sections. Here we only describe the deposit agents in the merchant service and the bank service. The structures of the deposit agents are shown in Figure 6.

7 Conclusion

We have described a framework for building an electronic currency system. We presented the overall structure of the system. We also introduced the basic techniques to achieve both atomicity and anonymity in an electronic currency system. Our framework is also protocol-independent in the sense that those Chaum-like electronic currency protocols can be incorporated into the framework. We are still in the process of designing recovery mechanisms to give a detailed specification of the redo_transaction procedure.

References

[1] O. Babaoglu and S. Toueg. Non-Blocking Atomic Commitment. In Sape Mullender, editor, *Distributed Systems, 2nd Edition*, pages 147–168. Addison-Wesley, 1993.

[2] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[3] D. Chaum. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Comm. of ACM*, 24(2):84–88, 1981.

[4] D. Chaum. Security without Identification: Transaction Systems to Make Big Brother Obsolete. *Communications of ACM*, 28(10):1030–1044, 1985.

[5] D. Chaum, A. Fiat, and M. Naor. Untraceable Eletronic Cash. In *Advances in Cryptology-Crypto88*, pages 319–327. Springer-Verlag, 1989.

[6] Niels Ferguson. Extensions of Single-term Coin. In *Advances in Cryptolog-Crypto'93.* Springer-Verlag, 1993.

[7] M. Franklin and M. Yung. Towards Provable Secure Efficient Electronic Cash. Technical Report TR CUCS-018-92, Columbia University, 1992.

[8] J. Gray and A. Reuter. *Transaction Processing.* Morgan-Kaufmann, 1994.

[9] V. Hadzilacos. On the Relationship Between the Atomic Commitment and Consensus Problems. In *Fault-Tolerant Distributed Computing*, pages 201–208. Springer-Verlag, 1990.

[10] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. In Sape Mullender, editor, *Distributed Systems, 2nd Edition*, pages 97–145. Addison-Wesley, 1993.

[11] B. Lampson. Atomic Transactions. In *Distributed Systems–Architecture and Implementation*, pages 246–265. Springer-Verlag, 1981.

[12] D. Lomet and M. Tuttle. Redo Recovery after System Crashes. In *Proceedings of the 21st International Conference on Very Large DataBases*, September 1995.

[13] G. Medvinsky and C. Neuman. NetCash: a Design for Practical Electronic Currency on the Internet. In *Proceedings of the first ACM Conference on Computer and Communications Security*, Nov 1993.

[14] M. Rabin. Digitalized signatures. In *Foundations of Secure Computation.* Academic Press, 1978.

[15] Lei Tang. Verifiable Transaction Atomicity for Electronic Payment Systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems.* IEEE Computer Society, May 1996.

Chrg-http: A Tool for Micropayments on the World Wide Web

Lei Tang * Steven Low †

Abstract

Chrg-http is a simple and secure protocol for electronic payments over the Internet, especially in an intranet environment. It is designed to support those micropayments (or more specific, electronic publishing, which have costs ranging from pennies to a few dollars. A widely used secure system Kerberos V5 has been incorporated into the http protocol. The security and authentication of a transactions is provided by Kerberos, without expensive public key cryptographic computations, or on-line processing through a centralized payment processing server, which is the case to most of the existing electronic payment systems on the World Wide Web. Our implementation is based on the billing model (or the subscription model). The simplicity of the model also helps to reduce the charging cost overhead.

1 Introduction

The World Wide Web (WWW) [4] has become a very popular facility for dissemination of information on the Internet. However, most of the information on the Internet is free. Intellectual property owners have little incentive to make valuable information accessible through the computer network. An electronic payment mechanism should be provided to facilitate monetary transactions among the customers and the potential merchants on the Internet. There exists several proposed payment systems based on the World Wide Web [3, 1, 2]. However, expensive public key cryptographic algorithms are widely used for the purpose of authentication and security. While these payment systems are ideal for those high-value transactions, they are not suitable for micropayments, in which most of the information goods have values ranging from pennies to a few dollars.

In this paper, we describe the design and implementation of a lightweight electronic payment protocol called the *chrg-http* protocol. It is intended for electronically publishing copyrighted goods such as journal papers or digital images on the Internet [6]. The protocol is based on the simplest charging model–the billing model (or the subscription model). Authentication and audit are two key issues in this model. A widely used authentication system Kerberos Version 5 [9] has been incorporated into the Cern httpd server and the NCSA Mosaic browser to provide authentication and secure communication between the browser and the httpd server. The kerberized http protocol is the chrg-http protocol. The chrg-http protocol is implemented by adding a new method called CGET into the http protocol. The kerberized Mosaic browser supports a new URL access method called "chrg-http", for connecting to the kerberized httpd server.

On the httpd server side, we add a module called *CGET handler* into the Cern httpd daemon. The CGET handler processes the CGET requests from the Mosaic browser, authenticates the customer's identity, encrypts the file requested by the customer, and sends it to the Mosaic browser. The CGET handler also logs access information for generating the bills which are sent to the customers at regular intervals.

On the client side, we add a module called the *chrg-http handler* into the NCSA Mosaic browser. The chrg-http handler translates a chrg-http request into a request including CGET as its method which is recognizable to the httpd server. It also authenticates the customer's identity to the CGET handler, decrypts the encrypted file, and passes the file to the browser for display.

Our presentation is organized as follows. First, we describe the project goals, the design decisions, and the reasons behind them. Second, we give a brief introduction to the Kerberos V5 system, we also describe how we incorporate the Kerberos authentica-

*GSIA, Carnegie Mellon University. Pittsburgh, PA 15213-3891. Email: ltang@cmu.edu. The work was done when the author was a summer intern in AT&T Bell Laboratories in the summer of 1994. The views and conclusions contained in this document are solely those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Carnegie Mellon University, or AT&T Bell Laboratories.

†Dept. of Electrical and Electronic Engineering, Univ of Melbourne, Victoria 3052, Australia. Formerly a member of technical staff in AT&T Bell Laboratories.

tion protocol into the http protocol. Then we describe our implementation of the chrg-http protocol on the World Wide Web. We will show how the protocol is incorporated into the NCSA Mosaic browser and the Cern httpd server.

2 Goals

The goal of our system is to allow micropayments such as electronic publishing through the World Wide Web, especially in an intranet environment. We try to demonstrate that the systems built solely for the purpose of authentication in distributed systems could be extended to handle electronic payments in a distributed environment, especially for those micropayments. Our goal is to reduce the charging cost overhead by adopting a simple billing structure and using the existing distributed authentication infrastructure for payment security and nonrepudiation. Any dispute regarding a transaction between a customer and a merchant could be arbitrated by the trusted party, the key distribution center of the authentication system.

Security
We use security to refer to the most important properties of an system for electronic payment: secrecy, authenticity, integrity, and nonrepudiation. Secrecy refers to denial of access to information by unauthorized principals (e.g., eavesdropper on the network). Authenticity refers to validating the source of a message; that is, the message was transmitted by a properly identified sender and is not a replay of a previously transmitted message. Integrity refers to assurance that a message was not modified accidentally or deliberately in transmit by replacement, insertion, or deletion. Since our system is based on the billing model, authenticity is our main concern.

Low costs
Our system is intended for electronic publishing. Most of the items for sale have cost ranging from pennies to several dollars. We design our protocol as cheap as possible so that the transaction cost is a small fraction of the cost of the purchased item. Our system reduces the charging overhead in a number of ways:

- We use a simple charging model (the billing model) to build our system. The simple accounting structure lowers the cost of development and maintenance, hence the charging overhead.

- The authentication mechanism provided by Kerberos is strong enough so that the cost of breaking the system is far more higher than that of purchasing the items directly from the merchant. Moreover, Kerberos provides a lightweigh authentication mechanism based on private key cryptographic algorithms (e.g. DES). It is much efficient compared with those protocols based on the public key cryptographic algorithms, such as the EIT Secure-http protocol and the Netscape HTTPS protocol. This reduces the computational overhead, and the corresponding public key certification and management cost, hence reduces the charging overhead.

- Kerberos are widely used by many institutions for system and network security. The kerberized browser and httpd server can be incorporated into a local environment smoothly. This reduces the administrational overhead. This also makes our system an ideal candidate for micropayments in an intranet environment.

Transparency
We incorporate our chrg-http protocol with the Mosaic browser. The charging mechanisms inside the protocol should be transparent to the customers.

Compatibility
Our implementation of the security enhanced World Wide Web should be compatible with the existing httpd servers and Mosaic browsers so that the customers who use our browser can still access those old httpd servers and the customers who access our httpd server using the old Mosaic browser can still get those files which are free from our httpd server. Instead of assigning a new port number to the new protocol, we use the same port number used by the http protocol.

3 Design of the chrg-http protocol

3.1 Charging models

We have to select a charging model before we design our protocol. There are several commonly used models for designing our protocols. They are the billing model (or the subscription model), the credit card model, the electronic check model, and the debit model.

In the credit card model, the customer sends his credit card number to the merchant through some pre-established secure channel between them. We did not use this model because the credit card transaction cost is high (standard charging costs levied by the credit card company). It is not suitable for

our system, which is intended for the low cost transactions such as electronic publishing of journal papers. The SEPP (Secure Electronic Payment) protocol proposed by IBM, Mastercard, and Netscape is such an example [1].

The debit model is a good candidate for low-cost transactions. In this model, a set of electronic payment service providers are established to transfer funds between the customer's account and the merchant's account. We do not adopt this model for complexity reason. In this model, not only does the payment service provider need to provide authentication mechanisms between the customer and the merchant, but it also needs to provide financial services (e.g., funds transfer, fund management, etc) for the customers and the merchants. Since our project should be finished in three months and building a central billing server from scratch is not an easy task, we did not use this model either. But we did design a set of "cheap" protocols based on this model described in [10].

In the electronic check model, the customer signs digital checks against his checking account. The recipient of the digital check could verify the validity of those checks through the public key certificate issued to the signer by a trusted party. The electronic check has the best scalability among all these models. However, a scalable public key management system is very difficult to implement. So it is not an ideal candidate for our system. The USC NetCheck is such an example [8, 7].

We use the billing model for simplicity reason. In the billing model, every customer registers with different merchants and obtains services (or goods) from the merchants. Whenever a customer and a merchant want to begin a transaction, they authenticate with each other. If the authentication succeeds, then the merchant logs the customer's identity and purchases items, sends a bill to the customer at regular intervals. Authentication and audit are two key issues in this model. Authentication guarantees that only the subscribed members of the merchant can access this service. Audit guarantees the preciseness of the bill sent to his customers by the merchant. We have implemented our chrg-http protocol based on this model. The implementation details are deferred to the next section.

3.2 The incorporation of http and Kerberos V5

Our chrg-http protocol is a combination of the http protocol and the Kerberos authentication protocol.

The Kerberos authentication system is a security system and has been adopted by many organizations as a solution to the network authentication problem. Kerberos provides evidences of a principal's identity. A principal is generally either a user or a particular service on some machine. A principal consists of three parts: the name of a user or a service, the instance either null or representing particular attributes of the user, and the realm representing the domain the user belonging to.

Kerberos principals obtains tickets for services from a special server known as the ticket-granting server (or key distribution center). A ticket contains assorted information identifying the principal, encrypted in the private key of the service. We briefly explain the chrg-http protocol as follows. The details of the Kerberos protocol V5 can be found in [9].

The customer sends the identity of the merchant to the Kerberos Key Distribution Center (called KDC) and then he is prompted for a password to prove his identity to the KDC. The KDC generates a ticket including the identity of the customer, the identity of the merchant, the customer's IP address, the timestamp, the expiration time, and a random session key K_{cm} which will later be used for secure communication between the customer and the merchant. This is all encrypted in a key K_{ma} known only to the KDC and the merchant. The KDC then sends the ticket, along with a copy of the session key K_{cm} and some additional information, back to the customer. The response is encrypted in the customer's private key K_{ca}, known only to the KDC and the customer. Once the customer receives the message from the KDC, he obtains and sends the ticket to the merchant. The merchant verifies the ticket and sends the encrypted file back. The merchant can also prove his identity to the customer using the mutual authentication mechanism provided by the Kerberos system.

Now we begin to describe how the Kerberos V5 protocol is incorporated with the http protocol [5]. We add a new method `CGET` into the method field of the http request structure. Besides the functions of obtaining a file for the Mosaic browser, `CGET` also completes a mutual authentication between the httpd server and the Mosaic browser using the protocol described above. We do not put the authentication tokens (Kerberos ticket) in the MIME header as suggested by the http protocol specification. If we did so, an adversary could launch an active attack on this scheme. The adversary could obtain the message header by eavesdropping the communication. Then he takes out the authentication tokens from the header and puts them into other messages, since there is no way to guarantee the integrity of the

message in the http protocol. Another reason is that the message format of the Kerberos ticket is not consistent with the specified http message format. Instead of putting the Kerberos ticket into the MIME header, we pass the socket connection between the browser and the httpd server to the Kerberos authentication routines. The httpd server and the Mosaic browser pass the Kerberos tickets through the TCP connection. After the authentication succeeds, the messages passed between the httpd server and the Mosaic browser are the same as the message format specified by the http protocol except that the content of the message is encrypted.

In order for the customer and the merchant to do monetary transaction through chrg-http, the merchant has to register a secret key in the customer's realm, or the customer has to register a secret key in the merchant's realm, or both the customer's realm and the merchant's realm has to share a secret and authenticate each other through the Kerberos cross-realm authentication mechanism.

Another inconsistency between the Kerberos protocol and the http protocol is the specification of the user identities. In the http protocol, the user identity is denoted by a Unix Id (i.e., a number). Two different users on two different operating systems may have the same user identity. In order to make sure that the identity of the user is unique, the Kerberos principal name should be used as the user's identity. This is another modification to the http protocol.

3.3 Comparison to other secure http protocols

The most related work to chrg-http is the Secure http protocol designed by EIT and the Secure Socket Layer (SSL) protocol designed by Netscape. The goal of both protocols is to extend the http protocol for secure communication and authentication although their approaches are slightly different. Public key cryptographic algorithms such as RSA, and public key based systems such as PEM and PKCS-7, are widely used to achieve secure communication and authentication. In order to run the secure http protocols, either a unique port number is assigned to accomplish the protocol in a different channel other than the channel used by the http protocol (SSL), or the specifications of HTML, URL, and http header format have to be modified (secure http). Although these protocols scale better than our chrg-http protocol, computationally they are more expensive than our protocol. Moreover, it is assumed that there exists a trusted public key certification center to certify every public key used in the protocols. The complexity of the public key management will increase the charging overhead of those systems.

The Netscape https protocol based on SSL is implemented on a separated communication channel. A unique TCP port number is assigned to the https protocol. We use the same communication channel used by the http protocol to implement our chrg-http protocol. We pass the socket connection between the browser and the httpd server to the Kerberos authentication routines. The additional communication channel is not necessary in our protocol.

4 Implementation

Our chrg-http protocol is implemented in C based on NCSA Mosaic version 2.4 and Cern httpd server version 1.0 plus Kerberos V beta 4.2. In this section, we first introduce the World Wide Web, give an overview of the components of the system. Then we describe how we incorporate our protocol with the Mosaic browser and the httpd server.

4.1 The World Wide Web

The fundamental mechanism underlying the Web is the HyperText Transfer Protocol (also called the HTTP protocol). The HTTP protocol is a stateless, object-oriented, and request/response protocol. A client establishes a connection with the server. The client sends a request to the server in the form of a request method, a universe resource locator(URL), and a protocol version, followed by a MIME-like message containing request modifiers, client information, and possible body content. The server responds with a status line (including its protocol version and a success or error code), followed by a MIME-like message containing server information and possible body content.

The Web server (here we mean Cern httpd server) and the Web client (here we mean the NCSA Mosaic browser) use the HTTP protocol described above to exchange information. Here is the general scenario of the Web currently in use: The Web server exports a hierarchical file space to the Web clients. The file space consists of documents of various types. The primary function of the Web clients is to acquire documents from this file space in response to user actions. The client and the server must also negotiate a document format which is acceptable to them both. The Web servers and clients communicate with each other over TCP/IP streams.

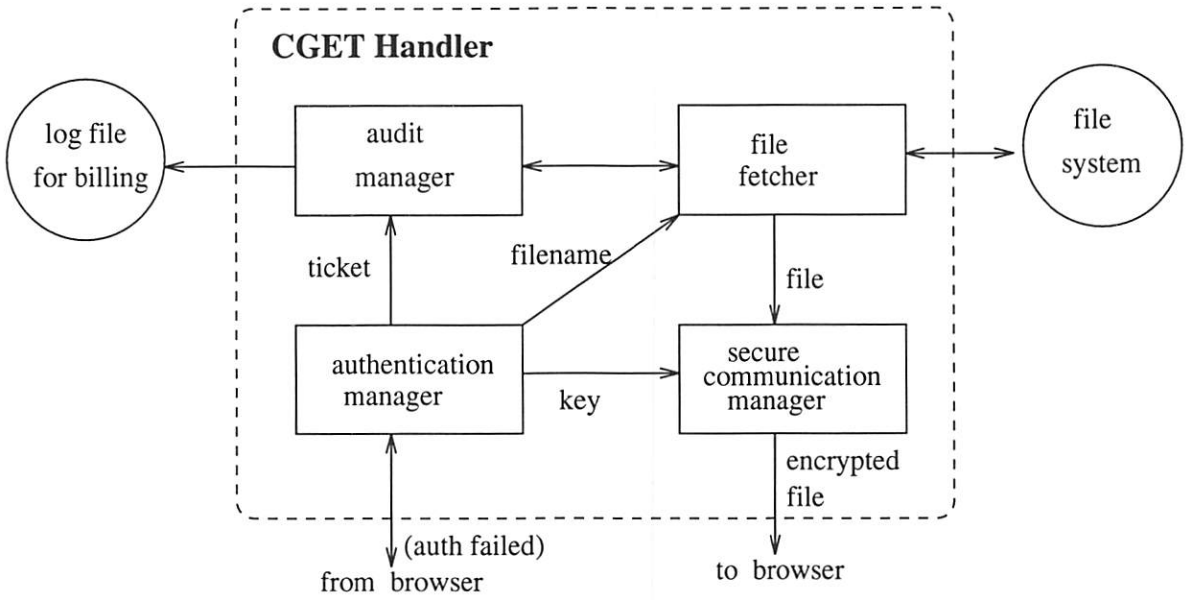


Figure 1: The structure of the CGET handler

4.2 The httpd server

We implement an independent module called the *CGET handler* and plug it into the Cern httpd server (version 1.0). The structure of the CGET handler is shown in Figure 1.

The CGET handler consists of four parts: the *authentication manager*, the *secure communication manager*, the *audit manager*, and the *file fetcher*. The authentication manager implements the Kerberos mutual authentication protocol between the Mosaic browser and the httpd server. If the authentication fails, the authentication manager sends an error message back to the Mosaic browser. If the authentication succeeds, the authentication manager caches the session key which will be used for secure communication between the httpd server and the Mosaic browser. The authentication manager also passes the name of the file to the file fetcher and the authentication token (ticket) to the audit manager. If the file fetcher finds the file in the local file system, the file is fetched and is passed to the secure communication manager. The secure communication manager encrypts the file using the cached session key and sends the encrypted file to the Mosaic browser. The way that the session key being cached is exactly the same as the way in Kerberos. In the mean time, the audit manager logs the following information to the log file: the customer identity, the IP address of the browser, the file name, the price, the timestamp and the authentication token. The log file will be used to generate the billing information for the customer and to handle possible dispute between the customer and the merchant.

In order to make sure that our new protocol is compatible with those web browsers and httpd servers in which our protocol is not implemented, we employ a new indirection method. Every copy of nonfree file is associated with a tag file in the same directory of the local file system or in some specified repository. The tag file serves two functions here: one is to serve as an access control list; another one is to provide price information and advertisement information. We illustrate our method with an example.

When a customer clicks on a hyperlink anchored by `http://www.cmu.edu/paper.ps`, the client opens a TCP connection with the httpd server on the machine `www.cmu.edu` and sends request `GET paper.ps` to the server. The server then checks if the tag file `paper.ps_chrg` exists or not. If the file exists, then the file `paper.ps_chrg`, together with an error message "402 Payment Request", are sent to the client. This file contains the price and advertisement information of the file `paper.ps` and a URL link `chrg-http://www.cmu.edu/paper.ps` to the file `paper.ps`. If the customer decides to purchase the document `paper.ps`, he clicks on the chrg-http URL link. The client translates the chrg-http URL link into the request `CGET paper.ps` and sends the request to the server. In the mean time, the customer is asked by a pop-up window to input his authentication information and the price of the item. The customer is also asked if the file `paper.ps` needs to be encrypted when the server sends the file to the

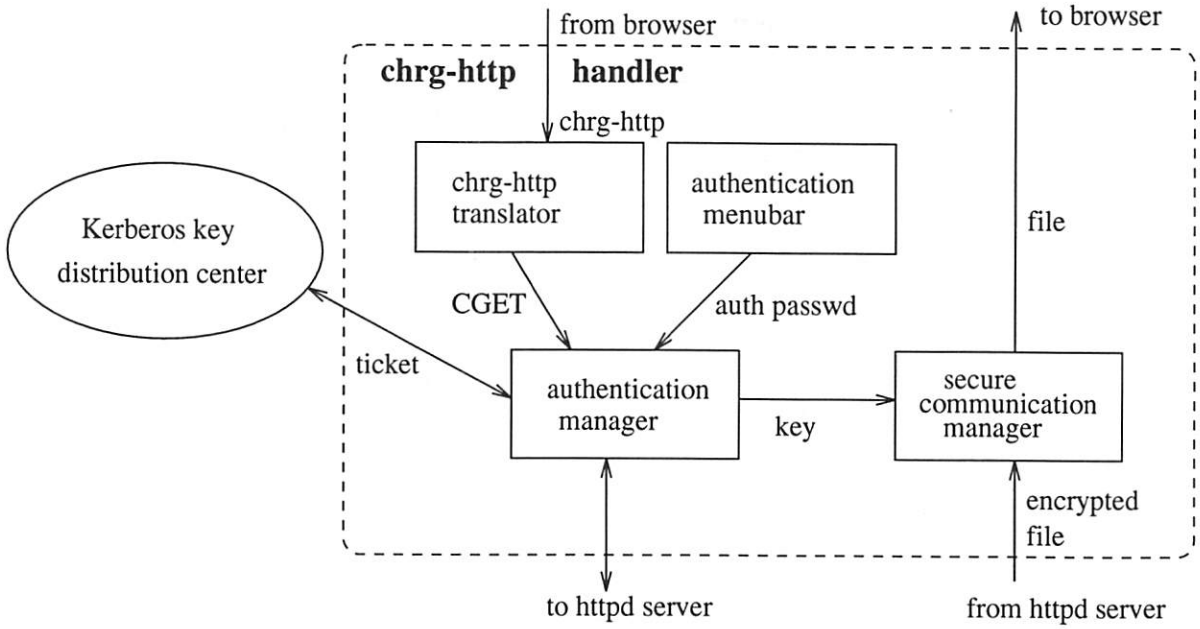


Figure 2: The structure of the chrg-http handler

client. When the server receives the request `CGET paper.ps`, it passes the request to the CGET handler.

One subtlety is how to handle the communication failure occurred during the execution of the chrg-http protocol. Since the http protocol is stateless, the httpd server does not know if the Mosaic browser receives the item it sends. If the CGET handler logs after the item is sent out from the server, the information may get lost due to network partitions. The customer does not receive the purchased item but his account is updated. This is unacceptable to the customer. We can not solve this problem in our chrg-http protocol since the Mosaic browser does not acknowledge the httpd server if it had received the items or not. This problem can only be resolved manually at the present time. If a customer complains that he has not received a purchased item, The administer of the server checks the log file and sends the item to the customer directly.

In case that the encryption of the purchased item is not desirable due to either political reasons (e.g., US government export regulation), or other technical reasons (e.g., the file includes multimedia data and the amount of data is huge, encryption of the data will degrade the performance of the server dramatically), we introduce a method called *one-time URL* to enhance the security of our protocol so that only those customers who have authenticated their identities to the server can get the purchased items. We illustrate our method with an example. Suppose that a customer wants to purchase a copy of video by clicking on a URL `chrg-http://www.cmu.edu/Jackson.mpeg` (the size of the file `Jackson.mpeg` is very big). If the customer passes the authentication, instead of encrypting the file, the server invents a random string R. The value of R is unpredictable. The server makes a symbolic link `R.mpeg` to `Jackson.mpeg`. Then the server sends URL `http://www.cmu.edu/R.mpeg` to the customer through the secure channel established by Kerberos and the customer can obtain the file from the server. Once the file `R.mpeg` is accessed, a daemon process (which is a part of the file handler) running on the server side unlinks the symbolic link.

4.3 The Web browser

Our implementation of the Web browser is based on the NCSA Mosaic version 2.4. We implement a module called the *chrg-http handler* and plug it into the Mosaic browser. The structure of the chrg-http handler is shown in Figure 2.

The chrg-http handler consists of four parts: the *authentication manager*, the *secure communication manager*, the *chrg-http translator*, and the *authentication menubar*. The chrg-http translator translates a
URL such as `chrg-http://www.cmu.edu/paper.ps` into `CGET paper.ps` and passes `CGET paper.ps` to the protocol handler inside the Mosaic browser. The protocol handler invokes the authentication man-

ager to authenticate mutually with the httpd server `www.cmu.edu`. The authentication manager selects the authentication method (right now the only choice is Kerberos V) through the authentication menubar and inputs the authentication password. Then the authentication manager obtains a ticket from the Kerberos key distribution center (KDC). The authentication manager caches a session key into the local file system (accessible only by the browser itself) used for decrypting files from the httpd server. It also implements the mutual authentication protocol with the httpd server based on the Kerberos ticket issued by KDC. If the authentication succeeds, the httpd server will send the encrypted file to the secure communication manager. The secure communication manager decrypts it and passes it to the browser and the procedure completes.

5 Conclusion

We have incorporated Kerberos V5 with the Cern-httpd server and NCSA Mosaic to provide a tool for electronic publication on the World Wide Web. This work was done in the summer of 1994. Since it is a three-month project, we do not have performance data to evaluate the efficiency of our system.

6 Acknowledgements

We are grateful to David Kristol and Nicholas Maxemchuck for their helps and discussions while this system was being designed and implemented. The first author was supported by a summer internship from AT&T Bell Labs.

References

[1] Secure Electronic Payment Protocol Specification. http://www.mastercard.com/Sepp/sepptoc.html, 1995.

[2] The Secure HyperText Transfer Protocol. http://www.eit.com/creations/s-http/draft-ietf-wts-shttp-00.txt, 1995.

[3] The SSL protocol. http://www.netscape.com/newsref/std/SSL.html, 1995.

[4] T. Berners-Lee, R. Cailliau, A. Luotonen, and H. Nielsen. The World-Wide Web. *Communication of the ACM*, August 1994.

[5] T. Berners-Lee, R. Fielding, and H. Frystyk Nielsen. Hypertext Transfer Protocol–HTTP/1.0. Internet Draft, March 1995.

[6] N. Maxemchuk. Electronic Document Distribution. *AT&T Technical Journal*, pages 73–80, September 1994.

[7] C. Neuman. Proxy-based authorization and accounting for distributed systems. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 283–291, May 1993.

[8] C. Neuman and G. Medvinsky. Requirements for Network Payment: The Netcheque Perspective. In *Proceedings of IEEE Compcon'95*, March 1995.

[9] C. Neuman and T. Ts'o. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications*, 32(9), September 1994.

[10] Lei Tang. A Set of Protocols for Micropayments in Distributed Systems. In *Proceedings of First USENIX Workshop on Electronic Commerce*, July 1995.

Building Systems That Flexibly Control Downloaded Executable Content

Trent Jaeger† Aviel D. Rubin‡ Atul Prakash†

Software Systems Research Lab†
EECS Department
University of Michigan
Ann Arbor, MI 48105
Emails: {jaegert|aprakash}@eecs.umich.edu

Security Research Group‡
Bellcore
445 South Street
Morristown, NJ 07960
rubin@bellcore.com

Abstract

Downloading executable content, which enables principals to run programs from remote sites, is a key technology in a number of emerging applications, including collaborative systems, electronic commerce, and web information services. However, the use of downloaded executable content also presents serious security problems because it enables remote principals to execute programs on behalf of the downloading principal. Unless downloaded executable content is properly controlled, a malicious remote principal may obtain unauthorized access to the downloading principal's resources. Current solutions either attempt to strictly limit the capabilities of downloaded content or require complete trust in the remote principal, so applications which require intermediate amounts of sharing, such as collaborative applications, cannot be constructed over insecure networks. In this paper, we describe an architecture that flexibly controls the access rights of downloaded content by: (1) authenticating content sources; (2) determining content access rights based on its source and the application that it is implementing; and (3) enforcing these access rights over a wide variety of objects and for the entire computation, even if external software is used. We describe the architecture in the context of an infrastructure for supporting collaborative applications.

1 Introduction

The ability to download executable content is emerging as a key technology in a number of applications, including collaborative systems, electronic commerce, and web information services. By downloading executable content on demand, systems can be built that provide better performance and fault tolerance than existing systems. Application performance can be improved because the program (i.e., executable content) can be downloaded to the location of the data (e.g., for a query) or the location of the user (e.g., for an interface-driven task). In addition, the fault tolerance of an application can be improved by reducing the client's dependency on the liveness of a specific server.

Recent distributed systems architectures, such as mobile agent and replicated process architectures, enable processes to download and run executable content. For example, consider the mobile agent architecture (also known as computational e-mail, command script architecture, and enabled mail) in Figure 1. First, a remote principal composes an agent by specifying a program. Through some mechanism (e.g., `http` or e-mail) the agent is downloaded to another principal. The downloading principal uses an agent interpreter process running on his machine to execute the mobile agent (number 2 in the figure). This process is owned by the downloading principal, so the agent is executed with his access rights. A malicious remote principal can use these access rights to: (1) read and write the downloading principal's private objects; (2) execute applications, such as `mail` or cryptographic software, to masquerade as the downloading principal to other users; and (3) read the password file on the downloading principal's machine. In addition, a remote principal may be spoofed into integrating malicious content from an attacker to mobile agent which would then give a third party access to the information provided above [5].

Current interpreters for executing downloaded content, such as Java-enabled Netscape, Java's ap-

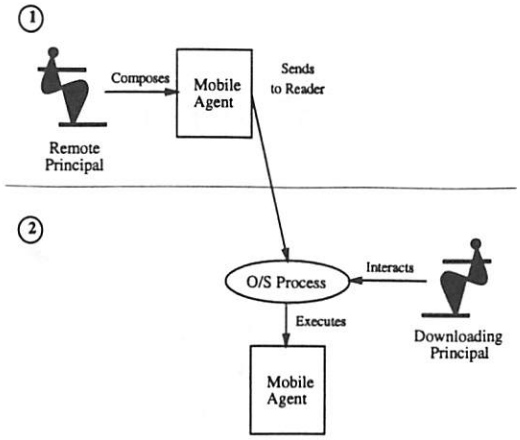


Figure 1: Mobile agent architecture

pletviewer [10], and Tcl's safe interpreter [3, 16], strictly limit the access rights of content to prevent these attacks. For example, content run using Java-enabled Netscape is prevented from performing any file system I/O and communicating with third parties (in theory anyway, see [5]). The Java appletviewer permits some access rights to be granted to content (as specified in the `~/.hotjava/properties` file). However, these rights must be shared with any remote principal whose content is downloaded. Therefore, these interpreters are not suitable for building an application where a downloading principal must grant rights to a specific remote principal.

Our goal is to flexibly control downloaded content, so applications which need to share resources with specific remote principals can be built. Flexible control of downloaded content means that the content's access rights can be set to any subset of the downloading principal's rights. Flexible control of downloaded content is difficult because: (1) to determine the access rights of downloaded content requires decisions at runtime and (2) to enforce these rights requires control of a wide variety of system objects. Content access rights depend on: (1) the trust in the remote principal who provided the downloaded content; (2) the access requirements of the application that the content is implementing; and (3) the state of that application. The state of the application and the access control requirements of the content are not known until runtime time, so a principal trusted to make access control decisions at runtime is necessary. However, users are typically not trusted to make such judgements. Also, executing content requires that access to a wide variety of system objects, such as files, sockets, and existing software, be controlled. In particular, control of existing software is not supported by interpreters because they are simply user processes. However, current operating systems also lack the necessary tools to effectively control external software executed by downloaded content.

In this paper, we describe an architecture for flexible control of downloaded content. This architecture enables downloading principals to execute applications that use content to share resources in a controlled manner to complete the application's goals. The architecture enables the access rights of downloaded content to be determined with little need for runtime access control specification by users. The architecture: (1) authenticates content sources; (2) determines content access rights based on its source and the application that it is implementing; and (3) enforces these access rights over a wide variety of objects and for the entire computation, even if external software is used. Cryptographic authentication identifies the source of content and also prevents some of the attacks that have plagued current interpreters, such as the DNS attack described in [5]. Also, in addition to the source, the application of the content is also used to determine the access rights of content. For example, we can remove the limitation that content can only communicate with ports at the server, by knowing the authorized remote principals in the application. We define an access control model for expressing the access rights of system objects and services for enforcing those rights. For example, the execution of external software is administered by a trusted interpreter, so we can control what software is being executed and limit the rights available to it.

Throughout the paper, we will assume a conventional protection model, where *principals* (e.g., users, groups, services, etc.) execute processes that perform *operations* (e.g., read, write, etc.) on *objects* (e.g., files, devices, etc.). The permissions of a principal to perform operations on objects are called the *access rights* of the principal. We call a set of access rights an *access control domain* or simply *domain*.

The paper is organized as follows. In Section 2, we detail the security requirements of some emerging applications. In Section 3, we define the problem of enforcing the access rights of an untrusted computation. In Section 4, we review related work. In Section 5, we present our system architecture. In Section 6, we define our access control model. In Section 7, we describe the architectural details and their implementation. In Section 8, we conclude the paper and present future work.

2 Example

At the University of Michigan, we are developing a system called the Upper Atmospheric Research Collaboratory (UARC) [6]. UARC provides its geographically-distributed users with a wide variety of applications which support collaborative analysis of atmospheric test data. For example, a test data viewer enables users located in the United States and Europe to share views of the atmospheric test data and jointly observe and annotate the views and the data. Most of the applications in UARC involve synchronous interaction among users.

Since collaborative applications have many common requirements, we are building an application-independent infrastructure called the Collaboratory Builders' Environment (CBE) [15]. The CBE provides services that are common to collaborative applications, such as replicated object management [22], multicast communication [11], and security. The goal of the security infrastructure is to provide the security services necessary to support a variety of collaborative applications.

UARC applications are to be implemented as downloaded executable content. When a user wants to use an application, the user downloads and executes the associated content on his local machine. Therefore, the user assumes the role of the downloading principal in the mobile agent architecture. In order to properly control the access rights of the user, the CBE infrastructure must be able to: (1) identify the source of the application and (2) assign that application appropriate access rights. For a UARC application, the set of UARC developers is the expected remote principal. Assigning the access rights is a difficult problem because a UARC application, such as the test data viewer, may need the rights to:

- Obtain test data from the remote UARC data server
- Communicate actions to the remote collaborators who share the view
- Read local setup files and environment variables for executing the application
- Read past analysis session files to replay them
- Store the analysis session for future replay
- Execute existing applications (e.g., editors, numerical analysis programs, etc.)

Therefore, we presume that an access control infrastructure must be able to control access to a variety of objects including file system objects, sockets, environment variables, applications, URLs, and network services.

Since the CBE supports collaborative applications, we must prevent collaborator actions from causing security problems. In the CBE, applications are supported by a replicated process architecture where each collaborator has: (1) a process on his local machine that is executing the application; (2) the application itself; (3) shared state which is to be consistently maintained by the application; and (4) private data that is unique to each collaborator. When a principal performs an action on the shared state, this action is multicast to each collaborator and executed at each site (i.e., if the principal is permitted to perform that action). Therefore, a replicated process architecture enables multiple principals to execute actions in a single process. Security problems arise because different principals have different rights in the collaboration. For example, some principals administer the collaboration, others participate by modifying the shared application state, others can make limited changes to the shared state, and some can only view the shared state. Since each collaborator has a copy of the application, some users may run a modified version of the application which could try to perform, either accidentally or maliciously, unauthorized actions. Also, attackers on the Internet may attempt to disrupt the collaboration by sending malicious messages, deleting messages, replaying old messages, and modifying messages in transit.

3 Problem Definition

The problem of flexibly controlling UARC applications that are supported by the CBE is to: (1) authenticate the source of the content; (2) determine the least privilege access rights for the content given its source; and (3) enforce those access rights throughout the execution of the content.

Authentication involves: (1) identifying the source of content; (2) verifying the integrity of content; and (3) ensuring that the content meets its freshness requirements. Authenticating the source of applications differs from authenticating the source of collaborator content. An application may be composed of components from several sources, so each of the sources needs to be identified to determine the appropriate access rights. Also, freshness is not a factor as long as the appropriate version of the application is obtained and its integrity is verified. Collaborator content is analogous to a message in a distributed computation,

so only one source is typically responsible for the content. However, the freshness of collaborator content must be verified because a replay of this content may cause the shared state of the collaborators to diverge.

The access rights of content depend on: (1) the trust of the downloading principal in the authenticated remote principals responsible for the content and (2) the purpose of the content. If a downloading principal trusts a remote principal with an access right, then the downloading principal can grant the remote principal's content that right. Clearly, this set of rights may be more than any particular content requires, and, in addition, there may be special cases where a downloading principal is willing to grant temporary access to a private file to complete some transaction. Therefore, information about the access requirements of content and the downloading principal's willingness to grant additional rights are needed to determine the rights actually required by content. Unfortunately, this information is not known until runtime and is typically difficult for end users to provide without being vulnerable to spoofing attacks.

The access rights assigned to content should be enforced for the content and any programs controlled by the content. Obviously, the access requests made directly by content must be controlled. In addition, any existing programs or network services that can be controlled by the content (i.e., can be provided input by the content directly or indirectly) must also be restricted to the same or fewer rights. In general, access to the following types of objects needs to be controlled:

- **Application Objects**: A process can read and write objects in its memory
- **File System**: A process can perform read, write, and execute operations
- **Applications**: A process can execute another application by creating a process for that application
- **Processes**: A process can communicate with another process via pipes or the file system
- **Environment**: A process can execute its environment's scripts and read and write its environment variables
- **Network Services**: A process can communicate with a service on the network using a socket
- **Universal Resource Locators (URLs)**: A process can download and read system objects by specifying a URL

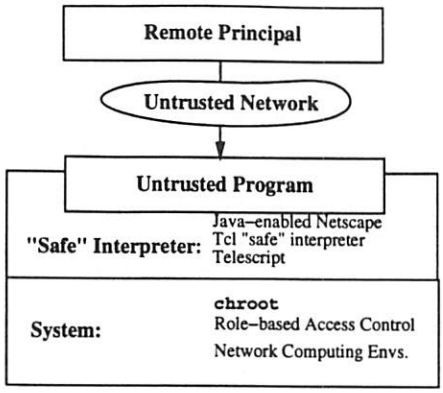


Figure 2: The current architecture for systems that utilize downloaded executable content

Operations on application objects must be authorized to ensure that the principal is permitted to modify the object. Access to file system objects should be limited to prevent unauthorized access to private objects. Also, access to objects that previously are available to trusted programs, such as environment variables and remote communication channels, must now also be controlled. These objects can be used by content to attack the downloading principal. For example, an attack on alpha version Java involves opening socket connections to an unauthorized third party to send information shared between the two legitimate principals [5].

4 Related Work

The current architecture for executing downloaded content is shown in Figure 2. In this architecture, executable content is downloaded to a principal who executes the program using one of a variety of "safe" interpreters, such as Java-enabled Netscape [10], Java's appletviewer, and Tcl's safe interpreter [3, 16]. To prevent attacks, these interpreters strictly limit the access rights of content. By default, all these interpreters only grant content (remote content in the case of Java) the right to communicate to only ports at the IP address of the content. The addition of read and write access rights to files is possible in the appletviewer by specifying those rights in the `~/.hotjava/properties` file. Unfortunately, the rights apply to all remote content. Also, these interpreters lack authentication, so all content must be assumed to have been downloaded from highly untrusted sources.

To grant rights to Tcl or Java content per remote principal currently requires the use of an in-

terpreter for trusted content or that custom applications be built. Interpreters for executing trusted content run content with the downloading principal's access rights, so access to objects not normally available to another principal, such as private files, are granted. We do not believe that content should be granted rights that another user would not normally have. On the other hand, custom applications require their own authentication, access control specification, and authorization infrastructure. Therefore, ad hoc security services need to be constructed which is an arduous and error-prone.

The Telescript engine [34] is another mobile agent interpreter. The Telescript engine differs from the interpreters described above in its use of *credentials* for authentication and *permits* for authorization. Credentials are cryptographic representations of the identity of the principal responsible for the content. Permits list the access rights of content. A permit can contain rights to another principal's (e.g., the downloading principal) resources. When content is downloaded, the downloading principal can deny rights that the content's permit grant, but this decision is ad hoc. Also, like other interpreters, the Telescript engine cannot control the execution of external software.

Operating systems can control the access rights of external software it executes, but current operating systems are not designed to flexibly restrict a principal's rights. For example, we showed in [12] that current file systems, such as Unix [24] and AFS [27], only provide limited mechanisms for a principal to dynamically restrict the access rights of one of his processes. Additionally, in Unix-based systems, the command `chroot` is available to limit the execution scope of a process to a file system subtree. However, `chroot` is cumbersome to use because of the need to transfer files to the restricted file system subtree, and, even worse, it cannot control remote communication by content. In fact, no current operating controls remote communication. Control of remote communication is typically provided by firewalls, but firewalls do not control access rights on a per process basis.

Recent research has yielded systems which provide support for defining limited access control domains, but it is not possible to generate a new domain at runtime. Role-based access control (RBAC) [1, 9, 33, 35] models permit a user to execute processes using different principals, called *roles*, which are associated with different access control domains. Thus, two processes run by the same user can have different access rights. However, to create these access control domains, most of these models require changes to the ACLs of all effected objects, so creating roles dynamically is prohibitively expensive. The Domain Type Enforcement (DTE) RBAC model uses the file system hierarchy to express domains more concisely. Therefore, we believe it is possible, from a performance perspective, to dynamically generate limited domains using DTE, but users and their processes are prevented from generating domains because DTE is used as a mandatory access control model. Other RBAC models permit evolution of access rights using rules [4, 17]. However, the rules, like the domains, must be specified in advance, but the rights of content depend on the goals of the content's application which are large in number and are often not known until runtime.

In [30], the DTE RBAC model is applied to controlling content. A domain that includes the objects needed by the browser to run and a "scratchpad," public directory are defined. While this permits an extension of current interpreter access domains, it still lacks the flexibility and per-content access control we desire. Also, control of remote communication is still lacking (however, IPC is controlled by DTE).

Another mechanism for granting limited rights to a process is delegation. Delegation is advantageous in that cryptographic credentials are used to represent the rights being delegated, but flexibility, standardization, and trust are problematic. Some systems, such as Taos [35], delegate rights via roles, so they suffer from the same flexibility limitations as RBAC. Kerberos version 5 [14, 29] provides a field for storing access control domains in the delegated credentials, so flexible delegation of rights over a variety of objects is possible. However, access rights must be transferred to servers in a language that the server can understand and enforce. Also, a trusted mechanism for transferring the rights to the server must be used. For example, a service controlled by the downloaded content cannot be given the responsibility to control file system objects.

5 Architecture

In the design of an architecture for controlling downloaded executable content, we make the following assumptions. First, we assume the existence of a public key infrastructure that be used to securely obtain the public key of any principal. Thus, any principal can verify the source and integrity of a message signed with a private key. Next, we assume that we can identify any I/O commands in the content language. This is necessary to control access to system objects. Next, we assume that the operating system has an unmodified trusted computing base, protects

process domains, and provides authentication of principals. This ensures that system software, such as cryptographic software, can be trusted, processes can only interact in controllable ways, and authentication of services is possible. Without trust in the operating system, it is not possible to build trusted applications that run on that operating system. A secure operating system, such as Trusted Mach [32], satisfies these requirements. Lastly, for applications involving three or more principals, we assume the existence of a secure group membership protocol, such as that described by Reiter [23], to ensure that all valid correctly-behaving principals share the same view of the application's group.

The following types of principals are involved in a downloaded content computation:

- **Downloading Principal**: The principal who downloads and executes the content
- **Remote Principals**: The principals responsible for the content
- **Application Developers**: Remote principals who provide content that implements applications that execute downloaded content from others
- **System Administrator**: A trusted principal who understands the structure of the downloading principal's system

The downloading principal trusts the system administrator and may have some trust in the remote principals (including the application developer). Thus, the downloading principal wants to grant only a subset of his access rights to a remote principal's content, and the system administrator is trusted to help the downloading principal define this subset. Note that the downloading principal may have different levels of trust (i.e., define different access control domains) in each remote principal. We assume that remote principals are trusted keep secrets from other remote principals with less privilege (e.g., private keys).

An architecture for flexible control of downloaded executable content is shown in Figure 3. This architecture consists of four levels: (1) local system services for access control; (2) a trusted, application-independent interpreter; (3) an application-specific interpreter (optional); and (4) an interpreter for executing downloaded content. The local system services provide operating systems services for controlling operations external to the interpreters. Also, system-specific information is made available for local system services (e.g., the file system structure).

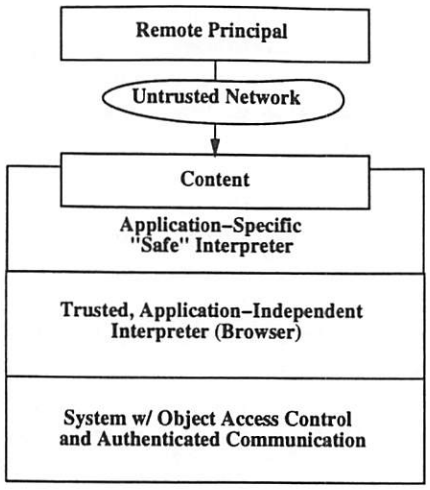


Figure 3: An architecture for flexible control of downloaded executable content: (a) system services control access outside the interpreter; (b) trusted, application-independent interpreters (*browsers*) control access to system objects by the less-trusted interpreters; (c) application-specific interpreters control access to application objects and determine the access rights of content within their domain; (d) downloaded content implements the actions of remote principals.

Trusted, application-independent interpreters (which we will refer to as *browsers* from here on) control access to system objects by the application-specific interpreters and downloaded content. An application-specific interpreter controls access to application objects and specifies the access rights of downloaded content within its limited domain.

The process for executing content using this architecture is shown in Figure 4. The remote principal sends a content message to the downloading principal which provides the content, the content type, and any authentication/encryption information. The browser receives this message and verifies the identity of the remote principal, the integrity of the content and content type, and the freshness of the message. If the verification succeeds, the browser determines which interpreter to execute the content. If the type refers to a known application-specific interpreter and the remote principal has the rights to execute content in that interpreter, then the content is run in that interpreter. The access rights available to the content are an intersection of: (1) the rights the downloading principal grants to the remote principal to execute the application; (2) the rights that the downloading principal grants to the application developer for the application; and (3) the rights that the application grants

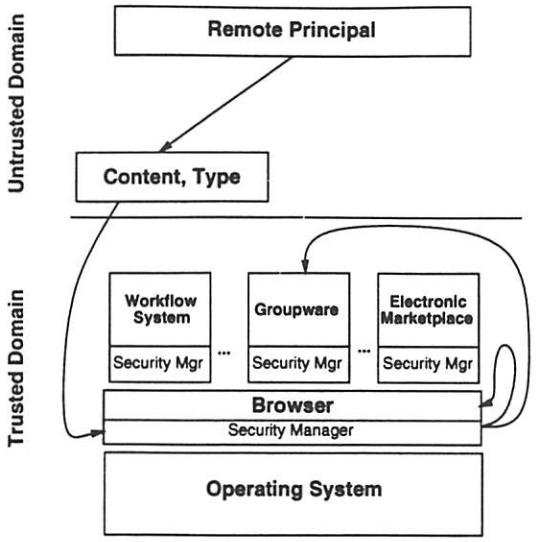


Figure 4: Content is downloaded to application-independent interpreter (called the *browser*) which authenticates the remote principal and finds the appropriate interpreter to execute the content. If the remote principal is authorized to execute content in that interpreter, then the content is executed.

to the remote principal. In addition, the application-specific interpreter has some leeway in transforming the access rights of content within its access control domain. This may result in either rights becoming available or unavailable from content, but the browser always enforces the intersection described above.

Content is executed as shown in Figure 5. After the browser authenticates the content and sends the content to be run in the appropriate application-specific interpreter as described above, the content is assigned to content interpreter based on the identity provided by authentication. This content interpreter has access rights consistent with the identity of the remote principal. Any controlled operation run by the content is authorized by the appropriate interpreter and if authorization is granted then the operation is executed. Operations are executed in the interpreter trusted with the operation. For example, file open is restricted to the browser. Also, the result returned to the content interpreter must not enable the content interpreter to gain additional rights. A read-only file handle is returned to the content interpreter in this case.

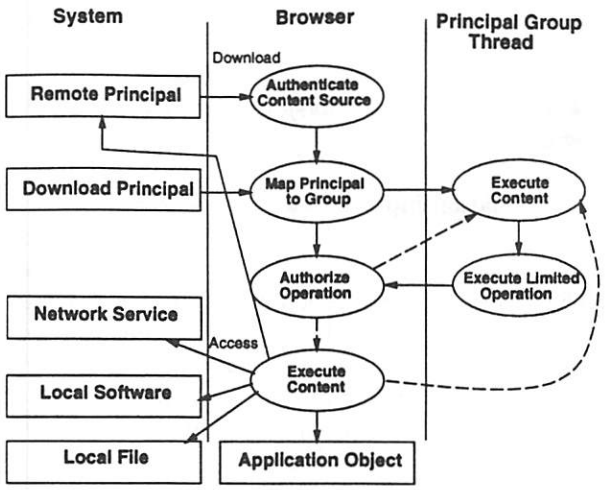


Figure 5: **Content Execution Protocol**: (1) Remote principal downloads content to browser; (2) Browser assigns content to principal group in application-specific interpreter (not shown); (3) content executes a limited operation which cause authorization in the browser; (4) if authorized the browser access system objects on behalf of content.

6 Access Control Model

Access control is the goal of this system, so we define an access control model for expressing the access control requirements of remote principals. This access control model uses the following concepts:

- **Definition 1**: A *principal group* is a set of principals with the same access rights.
- **Definition 2**: An *object group* is a set of objects which are grouped for expressing common access control requirements.
- **Definition 3**: *Access rights* of a principal group are the defined by two types of specifications:
  - **Domain Rights**: A tuple, $\{p\_group, allowed\_ops, object\_group\}$, which describes a set of operations which the principal group (*p_group*) can perform on an *object_group*.
  - **Exceptions**: A tuple, $\{p\_group, precluded\_ops, objectgroup'\}$, which describes a set of operations which the principal group (*p_group*) is precluded from performing on an *object_group'*.
- **Definition 4**: A *class operation* is a set of operations that a principal group can perform on

objects belonging to that class given authorization via object group rights.

- **Definition 5**: A *transform* moves an object from one object group to another.

The relationship between these concepts in the access control model is shown in Figure 6. Individual principals are aggregated into a *principal group* if they all have the same access rights. Objects are also aggregated into groups called *object groups* for expression of a common access control requirement. The *access rights* of a principal group are described by its domain rights and exceptions. *Domain rights* describe the rights permitted to the principal group, and *exceptions* describe rights which are precluded from the principal group. This permits access rights to be defined concisely for a large set of objects while permitting some objects in the group to override those rights. Thus, fewer object groups should be necessary. Note that exceptions take precedence in the authorization mechanism, so an operation is granted only if a domain right grants the operation and no exception exists that may preclude the operation.

The expression of object groups is fairly straightforward except in the case file system objects. Normally, object groups are simply unique names that refer to a set of objects. For example, suppose we permit content from any member of the UARC scientists group to communicate with any other member of the same group. We would express this rights as a domain right $\{scientists, communicate, scientists\_certs\}$. Definition of the UARC scientists group is simply a listing of their public key certificate identifiers (a more general specification is proposed in PolicyMaker [2]). File system objects are different because these objects already have access rights. We would like to use these existing rights to express a subset of the downloading principal's rights. We describe a set of file system objects by a path and a sharing type. A *path* indicates the domain in the file system hierarchy in which object group resides. For example, a path of $\sim$ indicates all objects in the downloading principal's home directory or any of its descendant directories. A *sharing type* indicates objects that are accessible to the same classes of principals. Examples of sharing types are:

- **All**: all objects that the downloading principal can perform the operations allowed (in the domain rights).
- **Intersection**: both the downloading principal and the remote principal objects can perform the operations allowed.

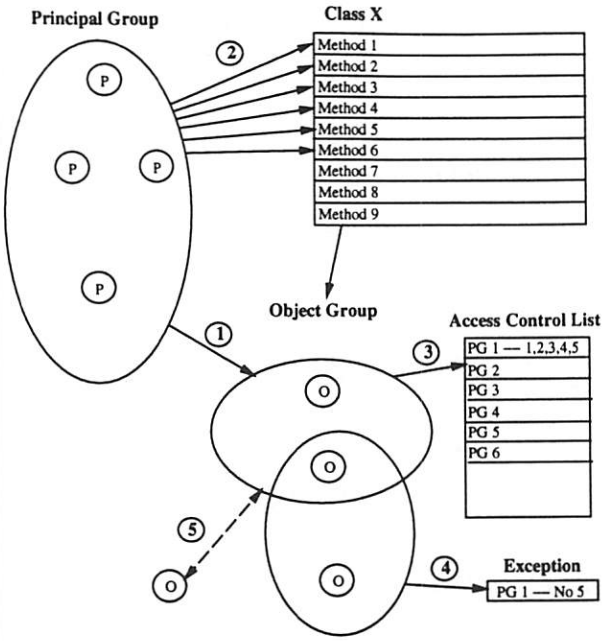


Figure 6: Access control relationships among entities: (1) **principal groups** can execute operations on objects that belong to classes (class operations); (2) principal groups can execute operations on objects in **object groups** (accessible objects); (3) **domain rights** specify a principal group's permissions to execute operations on an object group; (4) **exceptions** prevent operations from being executed on objects in an object group (even if a domain right is granted); and (5) **transforms** enable an object to join or leave a group.

- **Public**: The operations allowed for that are available to local users
- **Foreign**: The operations allowed for that are available to foreign users
- **None**: No objects
- **New**: Can only create, examine, and modify new objects in the domain

Using our access control model, the access rights for a UARC scientist using the UARC application described in the Example Section are (scientist is abbreviated as `sci`):

- $\{sci, communicate, uarc\_cert\}$
- $\{sci, communicate, scientists\_certs\}$
- $\{sci, r, uarc\_env\}$
- $\{sci, r, \sim /.uarc : public\}$

- except: $\{sci, rw, \sim /.uarc/prefs : all\}$
- $\{sci, rw, \sim /.uarc/sessions : new\}$
- $\{sci, rx, /usr/bin/num\_analysis : public\}$

UARC scientists' content can communicate with the UARC server and other UARC scientists. Also, environment variables for uarc can be used by the UARC scientists content. UARC scientists content can use downloading principal's file system to obtain public UARC setup information, except for the downloading principal's personal preferences. Also, existing, public analysis sessions in `~/.uarc/sessions` can be replayed given this specification. However, new sessions can only be saved into new files in that directory. Finally, this specification permits the content to execute a numerical analysis package.

The other access control model objects are used to help developers build their applications to manage access rights. The rights of principals to perform operations on classes of objects are defined using *class operations*. Class operations describe the set of operations that a principal group may ever perform. Class operations are strongly-typed (i.e., the types of the arguments must be well-defined). Also, casting of objects is restricted to ancestors or descendants in a class hierarchy to prevent unauthorized access by using a remotely different class's version of an operation with the same name. In addition, access control predicates on any argument in an operation can be added, which is useful for authorizing the execution of external software. Lastly, we define *mandatory class operations* as operations which can always be run (given that an object handle is available). These operations do not require authorization. All other controlled class operations (classes for which class operations are defined) require authorization.

The ability to transform access rights as the application state changes involves adding or removing objects from object groups. For example, when a downloading principal loads test data into a view, the window object should be loaded into an object group for shared windows. Thus, in the access control model, certain operations are associated with changes in object group membership, called *transforms*. The idea that operations modify the set of access rights, and that high-level specifications should be used to represent these access rights is adapted from Foley and Jacob [7]. However, our implementation specifies changes in rights rather than the complete set of rights. For example, the operation `x.load` returns a window object `y` upon load of test data `x`. In order to automatically add new windows to the shared window object group, developers specify that `y=x.load : swg.add(y)` where `swg` is the shared window object group. In order to execute this transform, the principal must have permission to run both `x.load` and `swg.add`, so restricting this operation such that only authorized principals can make access rights modifications is straightforward.

This model is influenced most strongly by the access control models of Hydra [36] and DTE [1]. Like Hydra, access control on the operations of abstract data types are possible, but access rights in our model are associated with principals rather than the content itself (procedures in Hydra). Therefore, management of rights is simpler and consistent with our applications. Like DTE, access rights information is aggregated to eliminate redundant specification. However, unlike DTE we use existing access control information to specify the domains. The sharing types permit us to implicitly restrict the access rights to a subset of the downloading principal's rights. Thus, verification that the domain is indeed a subset of the downloading principal's domain is not necessary. Also, we extend the application of DTE-like specification to non-file system objects.

7 Details and Implementation

The main tasks of the architecture are authenticating content, determining the access rights of content, and enforcing those access rights. Content must be authenticated to determining its source and verify its integrity and freshness. Determination of access rights involves combining trusts in the application developer and remote principal with the access rights implied by the application-specific interpreter's current state. Enforcement of these rights must be possible for the entire computation spawned by the content. This includes any external software or network services that are executed by the content. In this section, we detail these tasks and discuss their implementation.

We have developed a prototype architecture using Tcl version 7.5. We chose this version of Tcl because of its interpreter model and security model [16]. The interpreter model of Tcl 7.5 permits a hierarchy of interpreters to be constructed where one can be the master of another which is referred to as the slave. Therefore, we can build a hierarchy of interpreters where: (1) the browser is the master of the application-specific interpreters and (2) application-specific interpreters are masters of the content interpreters. In addition, Tcl 7.5 implements the "safe" interpreters of the style of Safe-Tcl [3]. Therefore, security is enforced by removing unsafe commands from

slave interpreters. However, some unsafe commands may be needed to complete the application, so Tcl provides a mechanism for masters to execute unsafe commands on behalf of slaves (called `alias`). Therefore, the browser implements commands that operate on system objects and the application-specific interpreter implements commands that operate on application objects. These unsafe actions cannot be added by the content interpreters, so they must use the commands in the masters and authorization is enforced.

7.1 Downloading Content

Content is downloaded in a *content message*. A content message has three purposes: (1) to provide the content; (2) to determine which interpreter should execute the content; and (3) to authorize the execution of the content in that interpreter. Therefore, a content message is defined as a quintuple $m = (r, c, t, a, s)$ where: (1) r is the identity of the remote principal; (2) c is the content; (3) t is the content type; (4) a is the content's authentication information; and (5) s is an optional session identifier (for computations involving multiple interactions). The content type is a MIME type of the form `browser/application` where `browser` refers to the fact that the content is controlled by the browser interpreter, and `application` identifies an application-specific interpreter in which the content will be executed. The `.mailcap` file is used to store this information. The `.mailcap` may be subject to denial-of-service attacks, so a secure (i.e., root-owned) `.mailcap` should be used. Therefore, the mapping of content type to application-specific interpreter is likely to be maintained by a system administrator.

The authentication information a is a double $a = (n, i)$ where n is a nonce and i is the integrity verification field. The nonce is a random value provided to prevent an attacker from replaying an old message. If an attacker could replay an old message, an attacker could modify the state of a transaction. We use a timestamp and a per-remote-principal counter as the nonce. The timestamp represents a recent, fresh session (i.e., no two sessions started by the same principal can use the same timestamp), and the counter indicates whether the message has been run in the session by this principal. i is the integrity verification field which is used to verify the integrity of the message. Using a public key algorithm, such as RSA [25] or DSA [19]), i is a digital signature of a message created from the concatenation of the remote principal name, content, type, and the nonce.

Some applications involve a number of interactions, so to improve the performance of the message authentication in these applications, symmetric key cryptography should be used. It is fairly straightforward for two principals to exchange a symmetric key given that the two principals have securely obtained each other's public keys (e.g., SSL protocol [8]). If symmetric cryptography is used, i is a message authentication code (MAC) computed with a hash function (e.g., SHA [18]) rather than a digital signature (the same message is used, however). We prefer using a hash function MAC over a DES CBC MIC [28] because a DES CBC MIC is not guaranteed to be collision-free.

In addition, some applications involve more than two principals and a number of interactions, so using symmetric cryptography for authentication becomes more complicated. In general, an n-principal computation requires $O(n^2)$ keys total and $O(n)$ encryptions per message because each pair of principals must share a secret in order to be certain of the identity of the sender. The fact that in many applications multiple principals have the same access rights can reduce the number of keys and encryptions somewhat. Members in a principal group can share a symmetric key, but a key is required for each pair of group and non-member. Therefore, if g is the number of principal groups and n is the number of principals overall, the maximum number of keys required is $g * (n - n/g) + g$ [1] and the number of encryptions per message is the number of non-members + 1 (for the group). Note that if the number of principal groups is small (i.e., limited by a known constant), which is normally the case in a synchronous collaboration, the number of keys remains bounded by $O(n)$. Therefore, as long as n is less than 100, symmetric cryptography is still preferred.

The browser procedure for authenticating an untrusted program is shown in Figure 7. This procedure receives pointers to the remote principal's name, the content, the content type, and the authentication information from the content message. A pointer to a session object is obtained if a session identifier is provided. A session object is a tuple, $s = (auth, members, counter, keys, t)$ where: (1) *auth* is the authentication type of the session (either `public` or `symmetric`); (2) *members* is the set of principals involved in the session; (3) *counter* is an array of message counters for each remote principal; (4) *keys* is a

[1] This the number of keys required if there are 2 or more group members in each group. If $g = n$ (each principal is in a unique group), then the number of keys required is fewer because many keys are redundant. An internal key for the group is not needed because each principal is a group, so the number of keys is reduced by n. Also, the remaining half of the keys are redundant because, an extra key is added for every pair of principals because keys are distributed between each group and non-member normally. Therefore, the number of keys is $(n * (n - n/n))/2 + n - n = n(n-1)/2$, as expected.

```
Authenticate(r c t a s)
   r is the remote principal name
   c is the content
   t is the content type
   a is the authentication information
   s is a session
   /* First, get the key for the principal from
   the session or use r's public key */
   If (s is null) then k = public_key(r)
   else k = key(s,r)
   /* Next, If session is unknown or r's counter = 0
   then use public key algorithm to compute y
   if ((s is null) || (s.counter[r] = 0)) then
      y = f_public(r, c, t, a.n)
   else y = f_s.auth(r, c, t, a.n)
   /* Finally, determine if y is the same as a.i for
   the expected counter value s.counter[r]
   If ((y != a.i) ||
      (s.counter[r] != a.n.counter)) then error(s,r)
   increment s.counter[r]
   return r
```

Figure 7: Authentication procedure in browser interpreter

mapping of principals to keys stored by the downloading principal; and (5) t is the content type. First, this procedure determines the key to use to authenticate the message using the remote principal's name and the *keys* map. If there is no session then r's public key is used. Next, the authentication function is determined from the session. If the session is null or this principal is new to the session the public key algorithm is used. The authentication value $f_{s.auth}(r, c, t, a)$ is compared to $a.i$ from the content message. If they match, then the message is authenticate to have been from r.

Once r's content is authenticated, the name of the application-specific interpreter for the content is retrieved from the `.mailcap` file using the content type t (hereafter called interpreter t). Next, we determine if the remote principal is permitted to execute content in interpreter t. If the remote principal is mapped to a principal group in interpreter t, then the content can be executed. This mapping is stored in file called `principal-groups.`t. This file must be write-protected. Once it is determined that the remote principal's content can be executed, then the appropriate keys can be distributed to the remote principal, if necessary. First, if there is no session object, then interpreter t is queried for its authentication type. If there is a session object, it is examined to determine if a keys for the principal group have been assigned. If not keys for the group and between the group and any non-members are generated. Also, the remote principal needs a key for each group it is not a member of.

7.2 Determining Access Rights

Once the browser has determined that r's content can be run in interpreter t, it is then necessary to determine the content's access rights. We first describe the ways in which access rights are expressed. The goal is to minimize the amount of specification required by the downloading principal, yet to still enforce least privilege access rights. Next, we describe how the rights are computed. The access rights of a remote principal's content in interpreter t active at state s are the intersection of: (1) the rights of interpreter t and (2) the rights of the principal group to which the remote principal is assigned in interpreter t at state s. Application-specific interpreters are responsible for deriving access rights within their domain, but the browser still enforces all system object accesses.

Application-specific interpreters are responsible for deriving the access rights of remote principals, but they lack information about either the exact names of system objects, such as the path names of files, or the identities of trusted remote principals. Therefore, logical objects are created to represent these system-dependent objects. Remote principals are represented by principal groups and system objects are represented by object groups, as defined in the access control model. Thus, the access rights of the interpreter t and any remote principals is based on the mapping of these principals to principal groups and the mapping of object groups to actual system objects.

We first describe how the access control domain of interpreter t is specified. Object groups are used to express the types of the objects to which interpreter t requests access. For example, suppose interpreter t implements the test data viewer application described in the Example Section. Thus, interpreter t requires access to the following object groups:

- UARC Server Socket
- Scientist Sockets
- Administrators Sockets
- UARC Environment Variables
- UARC System Files
- UARC Analysis Data
- Numerical Analysis

- Emacs Text Editor

Thus, the following rights are specified for interpreter t using our access control model (UARC developers are abbreviated by `dev`):

- $\{dev : t, communicate, uarc\_servers\}$
- $\{dev : t, communicate, uarc\_scientists\}$
- $\{dev : t, communicate, uarc\_admins\}$
- $\{dev : t, r, uarc\_environment\}$
- $\{dev : t, r, uarc\_system\}$
- $\{dev : t, rw, uarc\_data\}$
- $\{dev : t, rx, external\_sw\}$

The file `/usr/local/uarc/system/mapping.`t is used to define the mapping from object groups to system objects for interpreter t. The contents of this file should be based on analysis (e.g., use in system that can log process actions) of the interpreter t. The result is an access control domain for interpreter t which describes the objects which it is trusted to protect. Therefore, significant trust may be placed in an application-specific interpreter. We do not think this is unreasonable given that traditional applications are typically run with the users' complete trust. The browser and operating system must be able to enforce the specified rights to ensure that unauthorized rights cannot be obtained by interpreter t, however.

We specify the following mapping for interpreter t in the file `/usr/local/uarc/system/mapping.`t:

- $uarc\_servers := uarc\_server\_cert$
- $uarc\_scientists := scientists\_certs$
- $uarc\_admins := uarc\_admin\_certs$
- $uarc\_environment := uarc\_environment$
- $uarc\_system := /usr/local/uarc/system : public$
- $uarc\_data :=\sim /.uarc : public\}$
  except: $w, \sim /.uarc/system : public$
- $external\_sw := /usr/bin : public$
  except: $rx, /usr/bin/mail : all$

This mapping results in the following access rights for interpreter t:

- $\{dev : t, communicate, uarc\_server\_cert\}$
- $\{dev : t, communicate, scientists\_certs\}$
- $\{dev : t, communicate, uarc\_admin\_certs\}$
- $\{dev : t, r, uarc\_environment\}$
- $\{dev : t, r, /usr/local/uarc/system : public\}$
- $\{dev : t, rw, \sim /.uarc : public\}$
- except: $\{dev : t, w, \sim /.uarc/system : public\}$
- $\{dev : t, rx, /usr/bin : public\}$
- except: $\{dev : t, rx, /usr/bin/mail : all\}$

Thus, interpreter t can communicate with the UARC server, scientists, and the UARC administrators. Also, it can access UARC environment variables and the files in the `~/.uarc` directory. The `/usr/local/uarc/system` and `.uarc/system` directories contain the secure files needed by the browser to execute the application, so write access to these files is precluded from interpreter t. Also, in this specification, permission is granted for it to execute all software in `usr/bin` except the `mail` program. The mapping of local groups to objects is done using `~/.uarc/groups` (for all applications).

The access rights of the remote principals in interpreter t is specified by defining the rights for principal groups in the interpreter and assigning remote principals to principal groups. The access rights for a principal group are specified by the developers in terms of object groups, as described above for the interpreter itself. Suppose the UARC application has two principal groups: `lead scientists` and `scientists`. Lead scientists can save new analyses on the machines of all collaborators, run the numerical analysis package, and replay and annotate analyses. Regular scientists can only replay and annotate analyses on behalf of the downloading principal. Therefore, the application developer specifies the access rights of these principal groups as follows (not including the communication and environment rights) (lead scientists are abbreviated `lsci`:

- **Lead Scientists**
  - $\{lsci, rw, new\_analyses\}$
  - $\{lsci, rw, annotations\}$
  - $\{lsci, r, old\_analyses\}$
  - $\{lsci, rx, num\_analysis\}$
- **Scientists**
  - $\{sci, rw, annotations\}$

– $\{sci, r, old\_analyses\}$

The system administrator and/or the downloading principal defines a mapping between these object groups and a domain of real system objects. This permits the browser to describe the maximal domains of principal groups to the downloading principal, so he can choose a principal group for a remote principal. The mapping is specified in the file `object-group.`t in the write-protected ~/`.uarc/system` directory (or perhaps in the `/usr/local/uarc/system` directory for a default mapping). For example, UARC's old analyses are mapped to ~/`.uarc/analyses` directory (similarly for annotations). However, initially there are no new analyses. A principal must create a new analysis in the ~/`.uarc/analyses` directory to write to it. The numerical analysis package is mapped to `/usr/bin/num_analysis`, but note that the developers must know the browser's API for executing it in order to correctly initiate its execution. This is a problem because we do not want to evaluate arbitrary commands in the browser. Evaluation of arbitrary code in a trusted interpreter can result in undesirable modification of the trusted interpreter's security data, so this cannot be allowed. Therefore, the application-specific interpreter requests execution of the logical numerical analysis package object, and the browser maps the logical object to the actual system object.

Once the files are mapped to objects groups, the initial access rights of the principal groups are well-defined. For example, the rights below result for the two principals:

- **Lead Scientists**
  - $\{lsci, rw, \sim /.uarc/analyses : new\}$
  - $\{lsci, rw, \sim /.uarc/annotations : public\}$
  - $\{lsci, r, \sim /.uarc/analyses : public\}$
  - $\{lsci, rx, /usr/bin/num\_analysis : public\}$
- **Scientists**
  - $\{sci, rw, \sim /.uarc/annotations : public\}$
  - $\{sci, r, \sim /.uarc/analyses : public\}$

Now, the mapping from remote principals to principal groups can be defined in the write-protected file ~/`.uarc/system/principal-group.`t. Note that if consistency is a requirement of the application, then all collaborators must agree on the principal group assignment. This is where a secure agreement protocol is necessary.

The actual rights of the remote principals depend on the actions that have been taken by the downloading principal in interpreter t. The interpreter uses the access control model's transforms to modify the access rights of principal groups implicitly based on operations by the downloading principal. For example, if a downloading principal loads or joins an analysis, then the remote principals are granted the right to update the associated annotation file on the downloading principal's system. The transform is `x.load : uarc_annotations.add(x.annotations)` where `x` is an analysis.

Therefore, the actual access rights of a remote principal's content are the rights which are delegated by the application-specific interpreter on behalf of the downloading principal that are within both the access control domains of the remote principal's principal group and the interpreter itself. Therefore, if the interpreter is malicious it will only affect files in its domain. If only the content is malicious, then it can only affect the dynamically-generated domain. Therefore, authentication of application-specific interpreters is vital, to obtain assurance that the interpreter is being granted the proper trust. We use the BETSI protocol to authenticate application-specific interpreters [26].

We would prefer that the downloading principal delegate rights to the application-specific interpreter and remote principals using the browser because it is trusted. However, we want delegation to be implicit relative to some application rather than require the downloading principal to specify access rights at runtime. The problem is to get the downloading principal's content interpreter to run with the application-specific interpreter's transforms securely in the trusted browser. Because the application developer is not completely trusted, allowing this code to be run in the trusted browser is problematic. Further work is needed here.

Since the application-specific interpreter provides access control information to the browser, authorizes access to application objects, and transforms the access rights of principal groups, this interpreter will require some security infrastructure. The architecture provides a service to modify application-specific interpreters to add security infrastructure given the access control requirements of the interpreter. First, a call is made to collect the access control predicates for the operation. There is always one predicate to test whether the operation is permitted on the object (i.e., first argument). Next, a call to the authorization procedure (described below) is added after the access control of each controlled operation (class operation that is not mandatory, see the Access Control Model Section). Finally, the transform operation is added to the end of each operation with a transform specification. If there are multiple they are ordered as

```
operation(args)
   args is a list of operation arguments
   arg_types is a list of the types of args
   arg_ops is a list of the ops to be run on args

   arg_types = types(args)
   If operation is not mandatory
      then collect predicate(operation, first(args))
         into arg_ops
   Foreach arg in args
      collect access predicates(operation, arg)
         into arg_ops
   If (authorize args arg_types arg_ops)
      then results = execute(procedure, args)
   Foreach transform in transforms
      apply-trans(transform,args,results)
```

Figure 8: Generated operation with authorization and transformation calls

specified (order may be important). An example of a modified operation is shown in Figure 8. Finally, an operation is added for obtaining the access rights of any principal group.

Content interpreters for the principal groups can also be generated in advance. However, the content interpreter cannot have actual implementations of controlled functions, otherwise a malicious content provide could override them with versions that circumvent authorization, for example. Tcl provides a mechanism called **alias** which enables an interpreter to call a command in another interpreter. Therefore, aliases are provided for all controlled operations in the content interpreter, so trusted implementations of the operations can be executed in the browser (for accessing system objects) and the application-specific interpreter (for accessing application objects).

7.3 Enforcing Access Rights

Lastly, the architecture provides support for enforcing the access rights specified above while executing content and any external software and network services used by the content. The browser: (1) authorizes content operations; (2) provides safe implementations of these operations; (3) enables controlled execution of external software and network services.

The actual authorization procedure is shown in Figure 9. The access rights of content to system objects specified in the previous section are stored in the browser. For each **arg**, the browser determines the object group of the **arg** and **arg_type**. The browser

```
authorize(args arg_types arg_ops)
   For arg in args, arg_type in arg_types,
      ops in arg_ops
      Find object group for arg and arg_type
      For op in ops
         If domain rights grant op
            AND no exception precludes op on arg
            then continue
            else return FALSE
   return TRUE
```

Figure 9: Authorization procedure

then determines whether the object access rights permit all the operations in **arg_ops**. Developers are not required to specify the operations on arguments because when another procedure is called, the subsequent calls will be authorized as well. However, specifying argument operations are useful to catch bugs in the procedure call and to restrict the access rights of external software. Typically, the only operation that is authorized is the application of the procedure to the first argument, the object identifier (i.e., the first argument in an object-oriented method indicates the object for which the procedure is intended). To authorize an operation on an object, the browser must be able to find: (1) a domain right that grants permission to perform this operation on this object and (2) that no exception exists which precludes performing this operation on this object. Note that we had to create object identifiers for Tcl objects since Tcl is not object-oriented.

Once authorized, the browser can execute the procedure. A procedure consists of two kinds of code: (1) calls to other procedures and (2) accesses to controlled objects. A browser executes calls to existing procedures in the content interpreter using the ***slave*** **eval** ***args*** call in Tcl. This ensures that all calls to other controlled procedures will be authorized using the principal group rights. Operations on controlled objects within the procedure are assumed to be authorized if access to the procedure is authorized. These commands must be run in the browser because only the browser has access to the systems object. It is hard to automatically distinguish from existing procedures in Tcl because Tcl is not object-oriented. Therefore, these methods are generated manually at present. We expect that the use of an object-oriented language would enable automatic generation of these procedures.

The browser provides safe implementations of **open**, **socket**, **env**, and **exec** operations. All these

implementations are fairly straightforward except for **exec** which we describe in detail below. The other operations are implemented by a call to authorize which checks the content access rights prior to calling the actual operation. Since these commands are only available in the browser and are protected by the authorization operation, only authorized accesses to system objects are possible. An example is the implementation of the **open** command in [13] (called **safe_open**).

The browser also provides support for controlled execution of external software and network services. When an operation for executing executing external software, the browser must: (1) authorize this execution and prepares for a safe execution; (2) determine a limited access control domain for executing the software or service; (3) transfer the domain specification to the system responsible for controlling the software; and (4) execute the software. This is not a easy task because current operating systems do not support dynamic modification of a principals access rights nor do they control remote communication.

In general, software execution is authorized by determining if the remote principal (i.e., principal group) has execute rights for the first argument in the **exec** call. If so, then the **exec** operation executes the software using the current rights of the principal group (described further below).

If the software should be further restricted or if we want to be sure that the software can be executed successfully, then we suggest the creation of classes for software to represent this information. Software classes contain strongly-typed "constructors" for executing the software. Also, access control predicates can be assigned to the arguments in the call. These predicates should authorize operations required of arguments as well as the values of arguments. For example, the command **cat x | xterm -e** pipes file **x** into a new xterm and executes it. We may want to prevent an argument of **cat** from being a pipe. Also, we'd like to determine if we have the rights to read file **x**. In addition, the access control predicates specify the rights required to execute the software (excepting some start-up privileges which can be specified separately), so the execution of software can be limited more strictly than the content.

Once the software execution is authorized and the access rights have been determined, it is necessary to convert the rights to a form that can be enforced by the operating system. Unfortunately, current operating systems are not designed to permit a principal to run a process with limited access rights. We describe a technique for limiting the access rights of a service in the Distributed Computing Environment (DCE) [21]. DCE has the advantage that its authorization model is very flexible. DCE associates ACL managers with services to implement the authorization mechanism of those services. Therefore, each service can use its own ACL manager. We define an ACL manager for implementing our authorization mechanism which will be used by DCE's distributed file system (DFS).

We first define our DCE access control model. DCE uses an extended Unix-style access model where foreign users join the standard owner, groups, and others principal types. Modification of access rights involves changing: (1) the user identity (UUID); (2) the groups identities and operations: **group_obj**, **group**, and **foreign group**; and (3) the operations that are permitted to the general classes: **other_obj**, **foreign_other**, and **any_other**. In addition, if intersection rights are granted, the remote principal and the downloading principal must be stored to determine if both have a specific right. The modified DCE model consists of the following fields and values (for a domain right):

- **Root**: The root path name for the rights
- **User**: Either the downloading principal (sharing type = all), **nobody@local** (public or intersection with local principal), or **nobody@foreign** (foreign or intersection with foreign principal), or none (none or new)
- **Group_Obj**: None
- **Groups**: Intersection of groups shared by the downloading principal and a remote principal (intersection)
- **Users**: Set of principals which must all have the right (intersection)
- **Masks**: ACL types and permissions granted to that ACL type (for all ACL types)
- **Communications**: List of principals or principal groups

Exceptions are specified similarly, but masks imply the rights precluded. Thus, the new ACL manager's authorization mechanism determines if an operation on an object is permitted by a domain rights specification by: (1) finding a domain rights specification whose root is an ancestor the the object; (2) determining if any instance of an ACL type grants access to perform that operation given the masks. For example, if the operation is write, then ACL type mask must permit write operations, and the instance of that type must have write permission on the object.

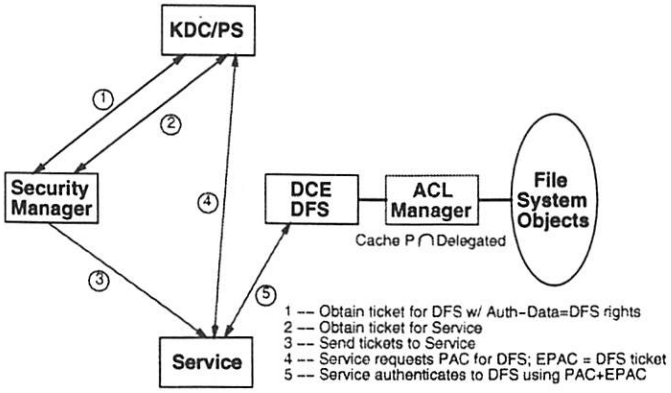


Figure 10: The protocol for a downloading principal to grant a service rights to his objects in DCE's Distributed File System (DFS)

DCE [21] utilizes an extended version of Kerberos version 5 tickets [14, 20] called Privilege Attribute Certificates (PACs) and further extensions to those tickets called Extended PACs (EPACs) for authentication. Kerberos documentation suggests that the `authorization-data` field be used to represent authorization information. However, DCE already uses that field for storing the UUID and groups for authentication, so we use the EPACs to store our authorization data. This enables a principal (the downloading principal via the browser) to grant rights to another principal (the service) which can be enforced by a third principal (the DFS). The protocol is shown in Figure 10. In this protocol, the browser obtains an EPAC for the DFS and a PAC for the service. Only the DFS can decrypt the EPAC and the EPAC is integrity-protected, so the DFS can verify that the EPAC is from the downloading principal. Therefore, the DFS can grant the downloading principal's rights to the service. Note that the service can only obtain the downloading principal's rights if the EPAC is presented to the DFS. The DFS ACL manager can interpret the rights, so it can authorize actions given those rights.

Note that using a DTE [1] makes the specification of rights much simpler, and enables the computation of intersections to be done at runtime. This is because access rights can be expressed more concisely. Therefore, we would prefer to use a DTE-style access control model in the future.

Limiting access to environment variables is easily implemented by restricting which variables are made available to the software in the `exec` call. Therefore, the transfer of application object rights and environment rights to the kernel is not necessary.

Control of other objects used by existing software, such as remote communications and network services, involves interaction between the kernel and network computing services. Therefore, the kernel should be able to interact with DCE as well to enforce strong authentication on all communications. As a proof of concept, this model of communication has been implemented in the Taos operating system [35]. In Taos, processes are associated with authenticators for proving the identity of the owner of a process to other processes. The Taos kernel is integrated with the authentication service, so all communication is authenticated. The ability to dynamically restrict the access rights of a Taos process is not possible given the current design. Similar services are also being developed for Trusted Mach [32] (upon which DTE is the access control model), so a nearly sufficient system environment is not that far off.

8 Future Work and Conclusions

We define an architecture that flexibly controls the access rights of downloaded executable content. In contrast to current interpreters: (1) strong authentication is used to verify actions by remote principals; (2) application-specific access control requirements are obtained which enable least privilege access to be enforced flexibly; and (3) these access rights can be enforced at both the application and system levels which enables comprehensive access control on the application.

By using this architecture, a variety of attacks can be avoided. For example, attacks in which data is sent to unauthorized principals (e.g., [31]) can be avoided because principals are authenticated and applications can identify the set of authorized remote principals. The use of services with poor security records, such as `sendmail`, can be avoided entirely. In addition, untrusted applications, such as ones that use `setuid`, can also be avoided by not granting rights to execute them.

Also, the architecture permits the use of rights not typically available in current interpreters. We provide specification models for developers to specify the access rights of principals in their applications and how rights can be transformed given user actions. The architecture provides a service to modify applications to integrate these security requirements with the application code. This enables developers to build systems which are less likely to contain security infrastructure bugs that can lead to attacks. Also, this enables application-specific requirements to be used to control

access. Thus, arbitrary restrictions, such as precluding communication with parties other than the originating host and preventing the execution of existing applications, can be replaced with more semantically-meaningful, application-specific access domains.

In the future, we plan to assist application developers in verifying the access rights granted to remote principals. That is, we want to be able to show application developers what a remote principal can do given a specific set of access rights. Thus, application developers can decide whether they have specified a satisfactory set of access control requirements.

References

[1] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. Practical domain and type enforcement for UNIX. In *IEEE Symposium on Security and Privacy*, pages 66–77, 1995.

[2] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1996.

[3] N. S. Borenstein. Email with a mind of its own: The Safe-Tcl language for enabled mail. In *ULPAA '94*, 1994. Available via anonymous ftp from ics.uci.edu in the file mrose/safe-tcl/safe-tcl.tar.Z.

[4] E. Born and H. Stiegler. Discretionary access control by means of usage conditions. *Computers & Security*, 13(5):437–450, 1994.

[5] D. Dean, E. Felten, and D. Wallach. Java security: From HotaJava to Netscape and beyond. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1996.

[6] R. Clauer et. al. A prototype upper atmospheric collaboratory (UARC). AGU Monograph: Visualization Techniques in Space and Atmospheric Sciences. In press.

[7] S. Foley and J. Jacob. Specifying security for CSCW systems. In *Proceedings of the 8th IEEE Computer Security Foundations Workshop*, pages 136–145, 1995.

[8] A. O. Freier, P. Karlton, and P. C. Kocher. The ssl protocol version 3.0. Internet Draft, March, 1996.

[9] M. Gasser and E. McDermott. An architecture for practical delegation in a distributed system. In *IEEE Symposium on Security and Privacy*, pages 20–30, 1990.

[10] J. Gosling and H. McGilton. The Java language environment: A white paper, 1995. Available at URL http://java.sun.com/whitePaper/java-whitepaper-1.html.

[11] R. Hall, A. Mathur, F. Jahanian, A. Prakash, and C. Rasmussen. Corona: A communication service for scalable, reliable group collaboration systems. In *Proceedings of the Sixth ACM Conference on Computer-Supported Cooperative Work*, November 1996. To appear.

[12] T. Jaeger and A. Prakash. Support for the file system security requirements of computational e-mail systems. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 1–9, 1994.

[13] T. Jaeger and A. Prakash. Implementation of a discretionary access control model for script-based systems. In *Proceedings of the 8th IEEE Computer Security Foundations Workshop*, pages 70–84, 1995.

[14] J. T. Kohl and B. C. Neuman. The Kerberos network authentication service. Internet RFC 1510, September, 1993.

[15] J. Lee, A. Prakash, and T. Jaeger. A software architecture to support open distributed collabroatories. In *Proceedings of the Sixth ACM Conference on Computer-Supported Cooperative Work*, November 1996. To appear.

[16] J. Levy and J. Ousterhout. Safe Tcl: A toolbox for constructing electronic meeting places. In *The First USENIX Workshop on Electronic Commerce*, 1995. To appear.

[17] I. Mohammed and D. M. Dilts. Design for dynamic user-role-based security. *Computers & Security*, 13(8):661–671, 1994.

[18] National Institute of Standards and Technology, U.S. Department of Commerce. *NIST FIPS PUB 186, Secure Hash Standard*, May 1993.

[19] National Institute of Standards and Technology, U.S. Department of Commerce. *NIST FIPS PUB 186, Digital Signature Standard*, May 1994.

[20] B. C. Neuman. Proxy-based authorization and accounting for distributed systems. In *International Conference on Distributed Computing Systems*, pages 283–291, 1993.

[21] Open Software Foundation. *Introduction to OSF DCE*, revision 1.0 edition, December 1992.

[22] A. Prakash and H. Shim. Distview: Support for building efficient collabroative applications using replicated objects. In *Proceedings of the Fifth ACM Conference on Computer-Supported Cooperative Work*, October 1994.

[23] M. Reiter. A secure group membership protocol. *IEEE Transactions on Software Engineering*, 22(1):31–42, January 1996.

[24] D. M. Ritchie and K. Thompson. The UNIX timesharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[25] R. Rivest, A. Shamir, and L. Adleman. On digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[26] A. D. Rubin. Trusted distribution of software over the internet. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, 1995.

[27] M. Satyanarayanan. Integrating security in a large distributed system. *ACM Transactions on Computer Systems*, 7(3):247–280, August 1989.

[28] B. Schneier. *Applied Cryptography.* Wiley & Sons, 1994.

[29] J. G. Steiner, B. C. Neuman, and J. J. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Usenix Conference*, pages 191–202, 1988.

[30] D. F. Sterne, T.V. Benzel, L. Badger, K. M. Walker, K. A. Oostendorp, D. L. Sherman, and M. J. Petkac. Browsing the web safely with Domain and Type Enforcement. In *IEEE Symposium on Security and Privacy*, 1996. Abstract for a 5-minute presentation.

[31] Computer Emergency Response Team. Java implementations can allow connections to an arbitrary host. CERT Advisory CA:96:05, 1996. Available at URL ftp://info.cert.org/pub/cert_advisories/CA-96.05.java_applet_security_mgr.

[32] Trusted Information Systems, Inc. *Trusted Mach System Architecture*, TIS TMACH Edoc-0001-94A edition, August 1994.

[33] S. T. Vinter. Extended discretionary access controls. In *IEEE Symposium on Security and Privacy*, pages 39–49, 1988.

[34] J. E. White. Telescript technology: The foundation for the electronic marketplace. General Magic White Paper.

[35] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, February 1994.

[36] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974.

Enclaves: Enabling Secure Collaboration over the Internet

Li Gong

SRI International
Computer Science Laboratory
333 Ravenswood Avenue
Menlo Park, California 94025, U.S.A.
(gong@csl.sri.com)

Abstract

The rapid expansion of the Internet means that users increasingly want to interact with each other. Due to the openness and unsecure nature of the net, users often have to rely on firewalls to protect their connections. Firewalls, however, make real-time interaction and collaboration more difficult. Firewalls are also complicated to configure and expensive to install and maintain, and are inaccessible to small home offices and mobile users.

The Enclaves approach is to transform user machines into "enclaves," which are protected from outside interference and attacks. Using Enclaves, a group of collaborators can dynamically form a secure virtual subnet within which to conduct their joint business.

This paper describes the design and implementation of the Enclaves toolkit, and some applications we have built using the toolkit.

1 Motivation

Most user interaction and collaboration over the Internet have been primarily via electronic mail. More recently, groupware applications including teleconferencing have become more widely deployed, especially within large organizations. However, due to the general lack of network security support provided by these systems, groupware has been used mostly within a local-area network environment or over private networks consisting of leased lines.

On the other hand, the openness and lack of security provisions in the Internet have made firewalls a popular instrument for companies and institutions to protect their Internet connections. Such an approach poses two major problems. First, against the widely perceived potentials of the Internet, firewalls make (real-time) user interaction and collaboration much more difficult, if not impossible, especially across organizational boundaries [4]. Second, firewalls cannot be universally applied. For example, small companies or individuals users may not be able to afford the installation and maintenance of firewalls. Moreover, in the case of home users and mobile users, providing an extra, physically separate firewall machine is not practical.

We advocate a method of providing the necessary security while still taking advantage of this broad Internet connectivity through the use of secure "enclaves," where users are protected from outside interference and resistant to security attacks. (Many users "own" their desk-top or lap-top machines, so in theory they can turn off unwanted network server daemons on these enclaves.) For example, it would be very desirable for a group of colleagues, who are in different physical locations and are connected through the Internet, to be able to form a secure virtual subnet within which to conduct their joint business. It is also more convenient and economical if the secure formation of these virtual subnets does not depend on firewall machines or extra hardware other than the user machines.

In a fictitious application scenario shown in Figure 1, a group of colleagues are connected via the Internet, where each member is protected by the local enclave (denoted by the "circle" around each member). Those in possession of the correct authentication token (the "rose") can securely join the group, which is organized by the group leader. One member is using a mobile computer, and another member is a notary service whose job is to take down notes of transactions occured within this session.

In this paper, we describe the design and implementation of a toolkit, called "Enclaves", which is a con-

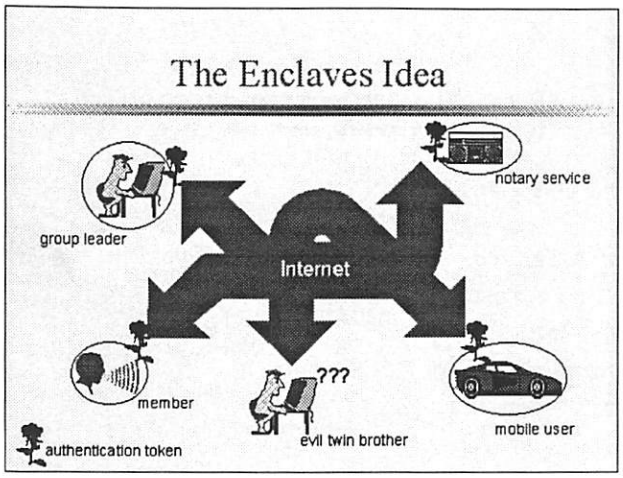


Figure 1: Using Enclaves

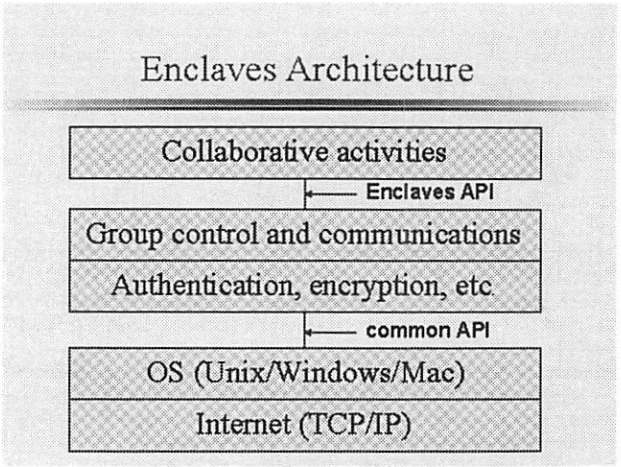


Figure 2: Layered architecture

crete demonstration of how to integrate security into user-level group-oriented applications. It shows that it is feasible for multiple users to collaborate closely, efficiently, and securely over the otherwise unsecure Internet. Our design practices should be useful to systems security architects and our toolkit should facilitate future constructions of secure, network-based collaborative applications.

2 Enclaves Design

In this section, we first describe the abstract system architecture of Enclaves. Then we explain the ideas behind this design and also compare it with prior related work.

2.1 System Architecture

Enclaves consists of subsystems layered on top of each other. Directly over the Internet and the operating system is a mechanism for authentication, which involves encryption primitives. Using authentication is the group management layer that handles group initiation, membership changes, and group dispersal. Built on top of this is a layer of abstraction for secure point-to-point communication and secure multicast over the Internet. Finally, on top of these are secure user-level group applications, an example of which is a facility for secure file sharing among group members.

As shown in Figure 2, the Enclaves toolkit depends on commonly available APIs, such as TCP/IP, and provides application builders with a layer of Enclaves API. This new API automatically provides mechanisms for user authentication, key distribution, secure group management, and secure multicast among group members.

The logical connections between a group of enclaves is as follows. The group leader occupies a special and important role in that all control flow (such as authentication and admission control) is mediated by the group leader. Any data flow that affects the entire group, such as modification to a shared file, is also mediated by the leader. Other non-crucial data can flow directly between group members to improve efficiency, after such channels are securely set up (again mediated by the group leader).

2.2 Design Rationale

Enclaves follows a design paradigm based on the following observations. First, user-level groups have different characteristics from process-level groups (such as in Isis [3] and Rampart [14]) and require different system-support mechanisms. In particular, group members have common interests but do not normally engage in identical activities.

Second, the object of sharing among group members should be the target of an application rather than the method or the medium used for the application. This is in contrast with many existing groupware systems [5], where sharing typically occurs too far away from the applications. For example, through simple replication, users may share a common text editor or the same X-window display. Sharing at such low levels has a number of problems. A major problem is security. To share a reasonably powerful editor (such as Emacs) or an X window where one can start a Unix

shell, all users become one in terms of access privileges and must be totally and mutually trustworthy. This is clearly not an secure or desirable way of sharing. Another problem is undesirable interference. Suppose an Emacs editor is shared among three users, and one user starts a search for a word. Because the editor is shared, the windows used by the other two users will be affected even though they have no interest in looking for this word (e.g., the other users could not type from their keyboards without having interference from the search command that is in effect).

Moreover, user groups need to be flexible in their security controls and must be able to be dynamically formed and dispersed. In today's commonly available technology, to let a user (who might work for a competing company) be part of a group requires setting up a user account and granting the user more access than he is entitled to, sometimes with serious security implications. In Enclaves, users can be admitted to or barred from a group by a simple control mechanism, and group members share exactly those resources that have been explicitly introduced into the group for sharing, not more and not less.

Enclaves runs completely in user space without needing root privilege, and does not need much infrastructural support (such as a multicast kernel).

2.3 Technical Approaches

Enclaves assumes that initial key assignment to group members is dealt elsewhere – certification authorities can be very useful for assigning keys. More specifically, each member is assumed to share a (possibly long term) symmetric key with the group leader. Authentication (for joining in a group session) uses DES-style shared-key cryptosystems. If we can assume that members have public-key capabilities, then Enclaves can use alternative protocols that provide very strong password protection [8].

Enclaves provides an abstract layer of secure multicast. For each group session, different keys are distributed to members for encrypting multicast data. The implementation of this layer is expected to change in future versions to better handle issues such as failure recovery and efficiency [9].

The Enclaves toolkit does not yet directly support remote objects. To facilitate a simple kind of secure remote object invocation, our approach is to define a platform-independent object manipulation language for each type of shared remote objects. Each local operation, if deemed to have global effect on a remote object, is then translated into this manipulation language before being multicast to the group. There are emerging softwares with built-in remote object capabilities and we will take advantage of them as they become widely available (except that we may have to retrofit these software with security features).

Finally, our design enforces the security of groups in Enclaves through the following mechanisms:

1. The group leader and members mutually authenticate each other via authentication tokens.
2. The session key is securely distributed to new members, and is securely revoked and updated whenever a member leaves the session.
3. Group communication is secured by encrypting network traffic.
4. Group members can invoke only those predefined procedures (or operations) on remote machines (or objects) and can do so only through the secure protocol interfaces.

2.4 Comparison with Related Work

There are related works on group security in the context of distributed computing systems. The Rampart system [14] functions on the fundamental concept of secure process groups that are also virtually synchronized. A major difference between Rampart and Enclaves is that the target application domains differ significantly. Rampart aims to be a toolkit for developing highly secure and fault-tolerant systems. As a result, its protocols are relatively complicated and expensive to run. Enclaves, on the other hand, deals with user-level groups that operate in a different fashion. For example, since users do not necessarily engage in identical activities, maintaining a virtual synchrony between group members is not meaningful and can only waste system resources and slow down the applications. Nevertheless, certain types of group activities in Enclaves may benefit from the stronger semantics guaranteed by the reliable and atomic multicast protocols.

Another related work is IP-multicast [6] and its security considerations [2]. Superficially, IP-multicast and Enclaves have a lot in common: they both form user groups over the Internet and facilitate group-oriented activities. However, the two have very distinctive characteristics in reality. The typical group formed using IP-multicast technology over the M-bone [7] is large in scale, loose in user interaction, and difficult to control in terms of security. In contrast, Enclaves seeks to support close user interaction and collaboration, good real-time response, and high quality

of security control. Moreover, Enclaves is extremely lightweight and runs directly over TCP/IP protocols. In comparison, the deployment of IP-multicast requires special routers or tunnels and sometimes the recompilation of operating system kernels to incorporate IP-multicast software.

The topic area popularly known as *groupware* or *computer-supported cooperative work* (CSCW), which includes collaborative systems for workflow management, shared whiteboard, teleconferencing, and shared editing (e.g., [11, 15]), is obviously related. A major difference is that these systems have not yet paid serious attention to security issues. A recent survey of the field [5] shows that only a tiny number of vendors discuss security at all or show concerns over privacy issues resulting from collaboration. The discussion often stops at how to provide password controlled access and encrypted electronic mail. Enclaves, on the other hand, is built on the basis of a secure group management layer and a secure multicast abstraction, and thus can provide very strong security guarantees.

A few systems that provide secure file sharing (such as [16]) have drawbacks, compared with Enclaves, in that they do not necessarily support real-time interaction; they also depend on good user behavior to maintain consistency of shared files. Moreover, they may have to share a real (rather than a virtual) file system.

Recently, there have been proposals to use firewalls (such as SunScreen from Sun Microsystems) to structure secure virtual networks over the open Internet. The idea is to use a pair of firewalls on each link between different parts of an organization. With automatic encryption and other support by the firewalls, machines behind these firewalls can function as if they are connected via a private network.

The secure subnet concept in the Enclaves system extends to individual users and machines, and thus can provide security functionality in the absence of firewalls, such as in the case of small companies, where maintaining firewalls is too expensive, or in the case of mobile users (e.g., using laptop computers) where a separate firewall is often unavailable. In this respect, Enclaves is closer to the concept of "joint-ventures", where partners with limited mutual trust must perform restricted collaborative activities [4]. The Enclaves system can also supplement firewall protection in that it is more user-friendly to configure corporate firewalls to support an acceptable level of protection policy while leaving certain fine-grained controls to individual users. In fact, by running Enclaves as firewalls and proxies, secure subnets can be formed in a similar fashion to using pairs of firewalls.

3 Toolkit Implementation

Enclaves 1.0 is developed on a Sun SPARC-IPX running SunOS 4.1.3, and has been tested on a SPARC-IPC and a SPARC-20 running Solaris 2.5, and on a 100-MHz PC 486 running the Slackware 2.3 distribution of Linux kernel 1.2.8. All the machines are connected to the Internet via Ethernets.

Enclaves 1.0 software is programmed using mostly the scripting language Tcl 7.3, its graphical companion Tk 3.6, and a TCP/IP extension of Tcl called Tcl-DP 3.2 [13]. In addition, there is a small amount of C and Unix Shell programming. Encryption is done with the DES in Cipher Block Chaining (CBC) mode.

3.1 Secure Group Management

The typical user-level group considered in Enclaves is a distributed group, with members located in different places. The group membership changes dynamically, and the group typically exists for a relatively short time. Because of the group's exclusive membership and the openness of the Internet, the first line of defense in establishing a group and maintaining this exclusiveness is to properly authenticate the group leader and members.

Users wishing to participate in shared activities are initially equipped with seed secrets (such as passwords or public-key certificates) which are registered with the group leader. The process of obtaining these can be facilitated via another layer of indirection of authentication. In fact, to make the toolkit easy to use for a wide range of users, we try hard to minimize the infrastructure needed to run Enclaves. Thus, in a very common mode of group operation, it is often sufficient that users can be authenticated to the group leader via (long-term or one-time) passwords that are prearranged by out-of-band methods.

The design for the protocols to authenticate group members to the group leader, and among themselves, follows established practice, so we do not dwell on these protocols except by giving one example, as shown in Table 1. The notation $\{x\}_K$ denotes x encrypted with K, and "," denotes concatenation. We use M to denote a group member and L to denote the group leader.

In message 1, the member requests joining the group. The leader checks the validity of the request message, and if satisfied, then checks the secure group policy to see if such a user can be admitted. If yes,

msg.1. M → L:	group_join, M_ID, {M_ID, L_ID, leader_host_name, port_number, group_join, Time}$_K$
msg.2. L → M:	group_admit, {L_ID, M_ID, leader_host_name, port_number, group_admit, group_key, Time}$_K$

Table 1: A simple authentication protocol

it returns message 2, which includes the group key for communication among group members. If no, the leader returns a message "authentication failed." Note that the checking of the group policy is not represented in the simple protocol description of Table 1, nor are the failure semantics. This protocol is implemented with a single RPC.

Synchronized clocks are not required to use Enclaves. If clocks may not be accurate, then random numbers (known as *nonces*) are used to prove freshness in a different, four-message authentication protocol shown in Table 2. Enclaves provides a random number generator, based on well-known algorithms [12]. In this situation, although three messages is the theoretical minimum for mutual authentication, the four-message protocol is simple to implement using two nested RPCs.

msg.1. M → L:	group_join, nonce_1
msg.2. L → M:	nonce_2
msg.3. M → L:	group_join, M_ID, {M_ID, L_ID, leader_host_name, port_number, group_join, nonce_2}$_K$
msg.4. L → M:	group_admit, {L_ID, M_ID, leader_host_name, port_number, group_admit, group_key, nonce_1}$_K$

Table 2: A nonce-based authentication protocol

In designing the protocols, we followed the robustness principle [1] and the specific format of fail-stop protocols [10]. Therefore, the protocols are somewhat less efficient than they can be; optimizations are discussed later in this paper. We do not discuss the security of these protocols except by pointing out that, in Enclaves, a user name is an arbitrary string with no spaces, and is usually bound to a secret key (e.g., a password). Once a user initiates or joins a group, his or her name is also bound to a fully qualified host name (such as *anchor.enclaves.com*) and a particular port number. This implies that a user cannot have more than one instance that is active within the same group.

When a user is admitted to the group, the leader arranges a state transfer so that the new member catches up with existing members. Currently, the group state transferred includes the group membership list and references of all shared resources such as text files.

There can be numerous types of groups, which range from totally open ones, to moderately controlled ones, and to ultra-secret ones [9]. These groups have vastly different admission policies. When a leader initiates a group, it has the option to decide upon the particular admission policy for that group. The most commonly used policy in Enclaves is what we call *leader controlled.* In such a group, the leader has the authority to invite users to join the group, decides the access-control policy, and is in charge of all group-wide activities. For example, it is the group leader who announces membership changes. To represent the policy, we use a simple access control list containing pairs of user names and their keys. The group leader can make policy changes by modifying this list "on-the-fly".

During the course of our development, we found it difficult to present the various group policies with a suitable user interface such that a user can easily visualize the implications of the policy and can conveniently change it while still maintain some coherent access control semantics. A leader-controlled group policy, the default policy, is one of the easiest to implement.

The procedure of admitting a new member is depicted in Figure 3. Note that if nonces are used, then an extra, nested RPC should be added. It would be reasonable to delay the state-transfer process till a later time, e.g., when the user explicitly requests for some state information.

When a member leaves the group, the leader initiates a group-key change protocol so that the departing user can no longer take part in group activities uninvited (using the residue group key). A key change may also be prompted when the group leader suspects a compromise somewhere. Such key updates are totally transparent to the users and in fact should be done periodically.

The group leader can invoke a program (or a protocol) to be run by a group member, and vice versa. Currently, no directly executable code is shipped across the network connecting the group members. Moreover, all such remotely executable programs are predefined, and security checking ensures that only these

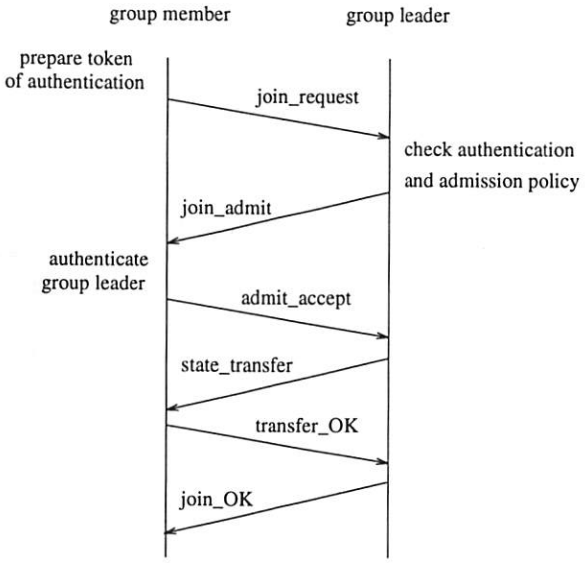


Figure 3: Admitting a new group member

programs can be invoked. Security checks are also performed on parameters when these programs are invoked.

A directory service, serving at a well-known port on a well-known machine, provides a listing of currently active groups. This service may also restrict information dissemination to certain Internet sites and to users with certain privileges. Such a service is straightforward to implement and we do not discuss further details. Of course, a group site may want to run unadvertised. In our on-going work, this service is integrated into the world-wide-web browser technology, so that the service is located at a well-known URL.

3.2 Secure Group Communication

The secure group communication abstraction provides both point-to-point communication and secure multipoint communication. Point-to-point messages are encrypted with a key shared by the two ends. Multicast messages are encrypted with the group key. For data flows, we use DES encryption in CBC mode, which provides data secrecy and integrity. Except for the first group-join request message, which has a segment of clear text noting the name of the requester, all messages are fully encrypted. (Anonymity can be achieved through other means that are not discussed here.) Following the fail-stop and the explicitness principles [1, 10], we make every message unique by including in the message the identities of the sender and of the recipient, a timestamp, and a sequence number, as follows: $\{$`Sender_ID, Recipient_ID, Sequence_Number, Timestamp, Message_Body`$\}_K$. Such a format ensures that a message cannot be reused successfully in a different place, time, or context. This rather conservative approach results in quite long messages even when the real message body is short. This can be optimized when the encryption key can identify the sender or the context cannot be misunderstood.

Currently, the secure multicast mechanism performs best-effort communication and does not provide additional semantics such as group-wide causality or atomicity [3]. For example, messages within the group are not necessarily causally or totally ordered.

There are a few reasons for this design choice. First, the weaker the multicast semantics, the faster the mechanism can be. Second, in a leader-controlled group, all crucial traffic is serialized at the leader site; therefore, no synchronization is necessary at lower levels. Moreover, users tend to tolerate failures more intelligently than processes. For example, suppose there is a temporary network link failure so that updates to a file cannot be communicated between a leader and a group member. A member can queue his changes locally and then "flush" them when the network becomes functional again. He can also push a button to refresh the file manually to pull in all the updates from the group leader. In other words, a weak low-level secure multicast semantics ensures efficiency, while upper-level mechanisms handle failures and exceptions. In the extreme case where a member has been cut off because of network partition, he can join the group again and instantly obtain the fresh group state. We believe that such an approach provides the best "economy" for general users.

When a leader becomes inactive (e.g., because of system crash or other failures), there is currently no mechanism to automatically reestablish the group with a new leader. It is not clear that such a mechanism is required or even desirable in user-level groups. For example, certain group tasks depend on a specific person as the group leader (e.g., a lead drafter of a contract), and it does not make sense for the group to continue in his absence. Certain types of user groups may benefit from automatic leader election, and this is an area for further study.

3.3 Performance Measurement

To see if our design and implementation are practical, we took some measurements on the time to complete crucial group-oriented tasks. We measured response time, that is, the time lapse between when a task is invoked and when it is completed.

The first task is initiating a group or session. This includes initializing group control data structures, reading in from disk an access control file, starting a server listening on a socket, and displaying a control panel.

The second task we measured is joining an existing group. This includes the member requesting joining by sending an encrypted message, and the group leader checking authentication and group admission policy and responding with an encrypted message to distribute a group key. These messages are followed by a state transfer phase, again via an encrypted channel. Note that this measurement obviously depends on the size of the current state, the complexity in checking the group admission policy, network latency, and speed of the crypto software.

Our measurement is done within a lightly loaded local-area network, a speeded-up software implementation of DES, a small ACL file (a dozen or so entries) and a minimal group state. Also note that the time it takes to notify all group members of this new member is not measured, because notification should be done asynchronously and some delay in this step is not crucial in a leader-controlled group.

The third task is leaving a group. In this signing-off process, the group member sends a request message, which is securely encrypted, and the leader verifies authentication and sends an acknowledgment. Note that to completely remove the member from the group, the leader must also clear all state information that depends on the member (such as locks on files) and execute the group-key change protocol. These various activities, however, are not measured since they should be done after the member is acknowledged. Their costs can vary a lot, depending on the size of the group and of its state information.

The measurements are represented in Table 3. It is not surprising that joining a group takes the longest time because it involves more interactions between the group member and the group leader.

	time in seconds
starting group	2.201
joining group	3.634
leaving group	0.569

Table 3: Measured response time of operations in a leader-controlled group (October 1995)

We must emphasize that all execution except encryption is done in a *scripting* and *interpreted* language (and in user space). Compiled code should have about 10 times speedup. Also, the prototype paid no special attention to code optimization. Given that these tasks are initiated by human users, who choose menu items, type in data (such as password), and click on buttons, a few seconds latency in each task is insignificant and tolerable. Other tasks, such as introducing a new shared file, updating a file, or posting a group message, take much less time.

We expect that these numbers will be significantly improved in our re-implementation of the Enclaves system, which has already begun. Apart from the obvious code optimization, there are a number of design choices to improve efficiency. One is to precompute a stream cipher using the group key and DES. This would save the time to initialize DES (e.g., the time to load in the key and arrange the key table), which is a very expensive part of encryption. A second method is to arrange session keys (in addition to the group key) between each member and the group leader, and then use these temporary keys in point-to-point encryption. The benefit is that such keys are fresh and can identify the sender. Thus, the message length can be reduced; for example, timestamps and sender-recipient identifications are no longer necessary.

4 Demonstration Applications

To demonstrate that Enclaves can facilitate the development of secure group applications, we now describe three simple applications we built using Enclaves 1.0. These are whiteboard, secure file sharing, and secure co-editing of text files.

4.1 Secure Whiteboard

A whiteboard for posting group messages is implemented on top of secure multicast. This board is a shared object in the sense that every message sent to the board appears on all whiteboards maintained by the group members, and messages sent by the same member do not arrive out of order. No other constraint is placed on the board. For example, a member can clear his board when it becomes full, and this clear operation does not affect other whiteboards.

In the example shown in Figure 4, a user is composing a draft response to a comment posted to the group whiteboard by a fellow group member.

The whiteboard serves as the communication medium in the absence of (or in addition to) voice and video connections between group members. The

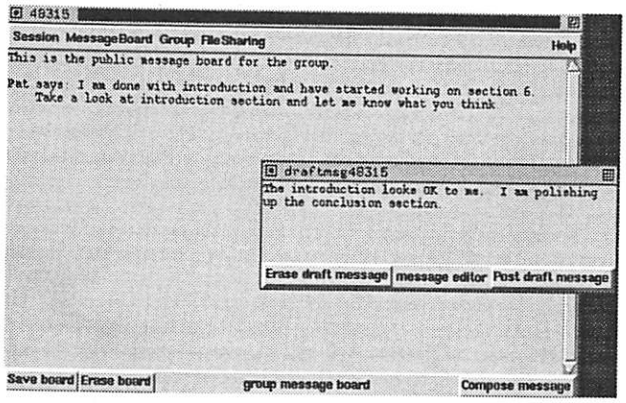


Figure 4: Composing a message for the whiteboard

content of the board can also be saved to a system log for auditing purposes.

4.2 Secure File Sharing

A rule followed in the Enclaves design is that, apart from group state information such as group key or group membership, each enclave is protected from other enclaves and the outside world in that nothing is shared until a group member explicitly introduces it into the group.

Consequently, when an Enclaves group is formed, the group shares a virtual file system space that is in most cases initially empty. This file space can be thought as something like `$GROUP_PATH/enclaves/` (where `GROUP_PATH` is a user-defined Unix shell variable) and may map to completely different positions within the file systems used by individual group members. In fact, two members could locally use entirely different types of file systems. When a member introduces a directory from his local file system into the group, the subtree becomes available to all group members. The availability of individual files depends on the protection bits on each file.

When a file is explicitly introduced into the group, a reference is entered into a shared-file list (i.e., the virtual file space in Enclaves) at every member's site. The file now is available for other group members (including the leader) to access. A member can click on the entry to access the file (e.g., view a gif file or edit a text file). Because the group leader coordinates all changes to shared files, the leader has the most up-to-date version. Thus, the open request is serviced by the leader, not by the member who originally introduced the file. This design choice can be changed, depending on the actual group structure.

4.3 Secure Co-Editing

For consistency reasons, even after opening a text file inside an editor, the member cannot edit the file (i.e., key strokes would have no effect) until he (or she) locks a region.

A region in this context is a continuous section of a file that contains zero or more characters. Locking is done by using the mouse to highlight a region of the file and then clicking the `lock region` button (or choosing from the menu bar). The lock request is sent to the server, which determines if it conflicts with existing locks, because two locked regions must not overlap. If there is no conflict, the server registers the lock and acknowledges it to the requesting member.

At this point, the locked region changes in color (say to blue) to visually signify that the lock has been granted. (We are considering non-color-based indicators for color-blind users.) Now the lock holder can edit within this region. Another member who also has the same file opened will see that this new region turns into red, signaling that it is locked by another group member, and will also see changes made by the other member reflected immediately in the local editor. When a member opens a file, all existing locks are reflected by color coding.

If a lock request is denied, the requestor is informed as to whose current locked region overlaps with the requested region. It is possible to assign priorities to users such that one with a higher priority can "grab" a locked region from one whose priority is lower. (Whether this is a desired group behavior is a separate question.) A lock can be explicitly released by clicking on the `unlock region` button. When a member exits from the editor, or leaves the group, his locks are automatically released.

To facilitate system-independent file editing, we have devised a simple language for file manipulation. We will not give all the details here, except for an example. In this language, inserting a character A is represented by a tuple of the form (`file_name, insert, A, cursor_index, region_name`). Here, the index is the relative distance from the start of the locked region.

This relative indexing has a few significant benefits. First, because the member has exclusively locked the region for writing, to maintain consistency, it is sufficient if the communication regarding this file between this member and the group member responsible for coordinating the file (in our case, the group leader) maintains a FIFO order. In other words, serialization is done via the group leader, so the regions can "float"

as a result of simultaneous editing, and yet there is still no need for a global causal or total ordering among file operations. Another advantage is that local batching becomes possible. For example, if a member is typing rapidly, the changes can be saved and then sent to other members in one batch. Of course batching can be done at lower levels, such as at the multicast level, although low-level batching is best directed by higher-level applications to obtain the best desirable effect.

One way to further improve efficiency is to design the object manipulation language to be as compact as possible. For example, a series of insertions can be replaced by a single insert command. Also, there is a lot of redundancy in commands such as the following, where the operation to delete a character from a shared file translates into a long command of the form (`file_name, delete, start_index, end_index, region_name`). If the redundancy in these commands is squeezed out, the amount of encryption is also reduced.

Based on this system-independent language, different group members can in theory use their own favorite editors (which of course must have been enhanced with this language capability).

Note that because a user has ultimate control over his own enclave, he can use methods outside of Enclaves to violate a lock and to update a shared file. However, any effect of such editing is strictly local because the group leader would not accept or propagate such an editing command.

So far we have focused on how a region within a single file can be locked and shared among group members. It is not difficult to generalize such a basic facility to sharing or locking entire files or even parts of the directory structure.

Finally, files can be saved to local disks and can also be signed with digital signatures. In fact, it is easy to implement "notary" services. A traditional object notary, as in a paper-based environment, receives submissions in the form of objects or files and returns (and maintain a record of) signed certificates. A session notary, on the other hand, can be implemented as a group member who takes notes of every change made to the shared files and commits these transaction records to a non-volatile storage with the proper digital signatures.

4.4 A Quick Review Tour

To review the overall workings of the demonstration applications built on Enclaves, let us suppose that Peter, Pat, and John are to add the final touches to a joint research paper, but John is traveling and Pat is working from home. Using today's typical technology, they would probably communicate via phone and electronic mail, and one of them would coordinate changes to the manuscript. This is inconvenient.

With Enclaves, things are quite different. Peter starts the control panel by typing `enclaves` at the shell prompt. At this point, Peter can either initiate a new session or join an existing session. Peter chooses to initiate a new session for finalizing the paper. By choosing `Start new session` from the `Session` menu, Peter is prompted for a few details, including the name of an access control file (ACL). In this scenario, the ACL file (called .access-control) contains two entries for John and Pat. This process is illustrated in Figure 5.

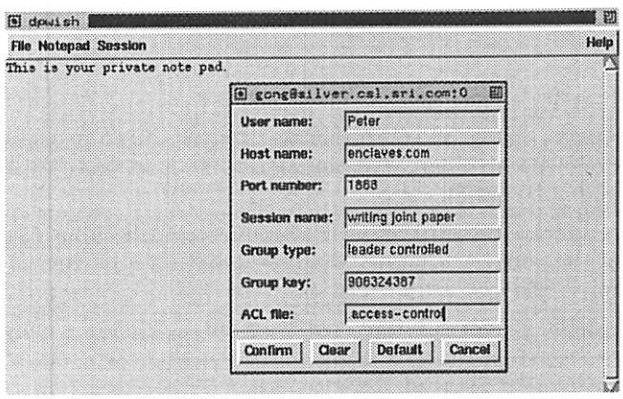


Figure 5: Starting up

By clicking the `Confirm` button, Peter starts a new session and gets a leader window for this session. Choosing the `FileSharing` menu (see Figure 6), he introduces into the group the file containing the joint paper to be shared among group members. Peter then locks the first few lines of the paper and starts refining the sentences. The locked region is automatically highlighted by a blue foreground and a white background.

A few moments later, Pat chooses to join the existing session. After giving the correct name and password, Pat is admitted to the group and gets a member window, which is similar to that of a group leader, except that it does not have leader functions such as access control. As Pat joins, a message is automatically posted on the group message board notifying this membership change. Figure 6 shows Peter's display at this point.

Pat can see from the shared-file list (under the `FileSharing` menu) that the joint paper is already inside the group and thus opens it. Immediately, he sees

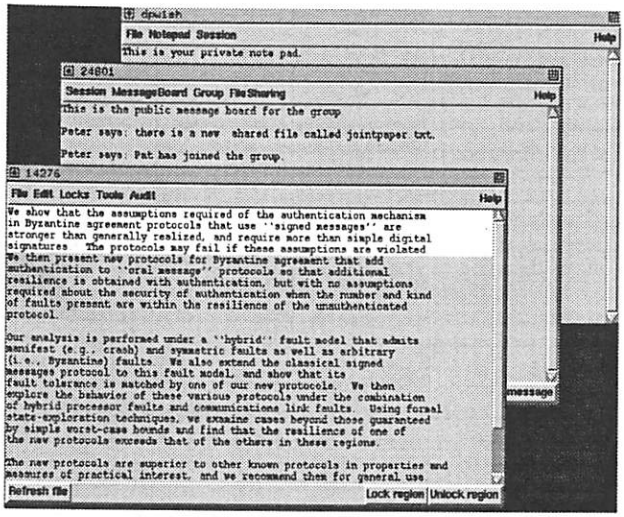


Figure 6: A group leader

that Peter has locked the first few lines and is working on them. He can instantly observe the changes as Peter makes them. Pat then locks the next paragraph and starts polishing.

Soon afterward, John also joins from the other end of the country via a local modem dial-up and then an Internet connection. The three of them communicate via the message board, working on the sections they choose. Figure 7 is a snapshot of Pat's editor window.

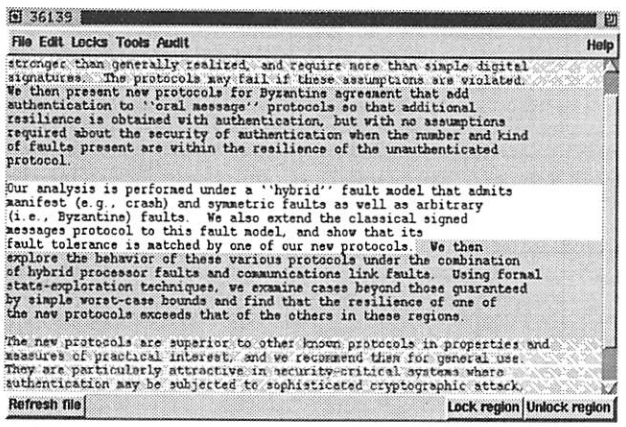


Figure 7: Co-editing

The top region is heavily shaded to indicate that it is currently locked by someone else (Peter, in this case). Similarly, the bottom region is locked by John. The region locked by Pat is the middle section with a solid white background. The rest of the paper – the gray regions – is currently unclaimed. (The locked and unlocked regions are color coded and will be more pleasant to see on a color display.) Each member can modify only a region locked by himself or herself and the group leader mediates the lock-granting process so that no two regions can overlap. These controls ensure that the replicated file copies maintain consistency.

When the completed paper is saved on disk and sent to printers, the group is simply dispersed. Throughout this session, all crucial communications over the Internet are encrypted. These communications include the authentication tokens and messages (passwords are never sent in the clear), messages posted on the group board, the text file, and any changes to it. An outsider cannot join the group for lack of a valid password, and cannot eavesdrop on useful information because of encryption. (We do not consider covert channels or traffic analysis.)

5 Summary and Ongoing Work

We have presented the design and implementation of Enclaves, which is an Internet-environment toolkit that makes it easier to build secure group-oriented applications. The toolkit contains mechanisms for dynamic group formation, secure group management, and secure group communication. Our prototype and preliminary performance measurements indicate that organizing secure, closely coupled user groups over the Internet is practical with currently available technology. Enclaves does not require privilege to install or run, and does not require a complicated access control mechanism.

To demonstrate the usability of the toolkit, we have built a secure file-sharing application that allows members of the same group to introduce files for sharing and co-editing within the group. The application allows concurrent editing via the concept of a locked region, which can be as small as containing zero or a single character, and a system-independent text file manipulation language.

We are currently developing version 1.1 to use Tcl7.5/Tk4.1, which are supported on Unix platforms as well as on Microsoft Windows and MacOS machines. One of the goals is to reduce platform dependent codes, which are mostly related to cryptographic operations. Parallel to this effort, we are developing Enclaves 2.0 from scratch, where integrating with web technology is a major objective. Another priority item is to incorporate secure audio and video into Enclaves, which will greatly improve coordination among group members. We are in the process of making the API and the code available.

Acknowledgements

This work is supported by SRI's internal research and development program. I am grateful for managerial support from the Engineering Research Group. Many thanks to Sam Owre for helping me out of various implementation traps, and to Don Nielson and Peter Neumann for technical and editorial comments.

References

[1] R.J. Anderson. Why Cryptosystems Fail. *Communications of the ACM*, 37(11):32–40, November 1994.

[2] A.J. Ballardie and J. Crowcroft. Multicast-Specific Security Threats and Counter-Measures. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, San Diego, California, February 1995.

[3] K.P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):37–53/103, December 1993.

[4] W.R. Cheswick and S.M. Bellovin. *Firewalls and Internet Security.* Addison-Wesley, 1994.

[5] D. Coleman and R. Khanna, editors. *Groupware Technology and Applications.* Prentice-Hall, Upper Saddle River, New Jersey, 1995.

[6] S. Deering. Host Extensions for IP Multicasting. Request for Comments 1112, Internet Network Working Group, August 1989.

[7] H. Eriksson. MBONE: The Multicast Backbone. *Communications of the ACM*, 37(8):54–60, August 1994.

[8] L. Gong. Optimal Authentication Protocols Resistant to Password Guessing Attacks. In *Proceedings of the 8th IEEE Computer Security Foundations Workshop*, pages 24–29, County Kerry, Ireland, June 1995.

[9] L. Gong and N. Shacham. Multicast Security and Its Extension to a Mobile Environment. *ACM-Baltzer Journal of Wireless Networks*, 1(3):281–295, October 1995.

[10] L. Gong and P. Syverson. Fail-Stop Protocols: An Approach to Designing Secure Protocols. In *Proceedings of the 5th IFIP Working Conference on Dependable Computing for Critical Applications*, Dependable Computing and Fault-Tolerant Systems, pages 44–55, Urbana-Champaign, Illinois, September 1995. Springer-Verlag.

[11] M. Knister and A. Prakash. Issues in the Design of a Toolkit for Supporting Multiple Group Editors. *Computing Systems*, 6(2):135–166, Spring 1993.

[12] D.E. Knuth. *The Art of Computer Programming, Vol.2: Seminumerical Algorithms.* Addison-Wesley, Reading, Massachusetts, 1969. Revised edition.

[13] J.K. Ousterhout. *Tcl and the Tk Toolkit.* Addison Wesley, Menlo Park, California, 1994.

[14] M. Reiter. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 68–80, Fairfax, Virginia, November 1994.

[15] M. Roseman and S. Greenberg. Building Real Time Groupware with GroupKit, a Groupware Toolkit. *ACM Transactions on Computer Human Interaction*, 1996. To appear.

[16] T. Takahashi, A. Shimbo, and M. Murota. File-Based Network Collaboration System. In *Proceedings of the 5th USENIX UNIX Security Symposium*, pages 95–104, Salt Lake City, Utah, June 1995.

Public Key Distribution with Secure DNS

James M. Galvin <galvin@commerce.net>
CommerceNet, PO Box 220, Glenwood, MD 21738

Abstract

Recently, many protocols in the Internet are proposing the use of public key cryptography in support of integrity and authentication security services. However, each of these protocols lacks a globally available public key distribution and management system. A secure version of the Domain Name System (DNS) is being developed which, conveniently, provides an infrastructure ideally suited for the distribution and management of public keys. We propose how this infrastructure of the secure DNS could be exploited by today's users of the Internet to distribute and manage their personal public keys.

1. Introduction

The use of public key cryptography in the Internet was first proposed by Privacy Enhanced Mail (PEM) [1,2,3,4]. PEM acknowledged the lack of a global key distribution and management system and, therefore, included an ad hoc mechanism for the distribution of public keys (embedded in X.509 certificates). Ideally, the X.500 Directory would have been become a technology of choice in the Internet, which would have provided a solution to the key (certificate) distribution and management problem. However, global deployment of the X.500 Directory has been problematic.

Recently, other protocols in the Internet are proposing the use of public key cryptography in support of integrity and authentication security services. However, similar to PEM, each of these protocols is in need of a globally available public key distribution and management system. In order to avoid the proliferation of ad hoc solutions it is essential that a single, unifying, integrated solution be proposed and deployed.

Fortunately, a recent protocol includes the specification of a global infrastructure that could be used to distribute and manage public keys for other protocols: the secure Domain Name System (DNS) [9]. As of this writing, it has been submitted for consideration as a Proposed Internet Standard. It is an enhancement of the DNS [5,6,7,8], an existing global infrastructure.

A global infrastructure for public key distribution and management must include a solution for the following objectives.

global availability - This is self-evident since a global infrastructure is not useful unless it is globally available. However, a critical aspect of global availability is scaleability; it must be possible to globally deploy a solution.

globally unique and unambiguous names - Since users prefer to be known by their name as opposed to their public key (a value that appears to be little more than a random sequence of bytes when displayed), each name must be globally unique and unambiguous. If the names are not unique and unambiguous, then the two users who share the same name would be indistinguishable by other users.

real-time access to public keys - Although other access methods are useful in some contexts, the real-time availability of public keys ensures that the global infrastructure is available to the broadest possible community.

cryptographically verifiable bindings between names and public keys - The introduction of names requires a mechanism for binding the names to public keys. The binding must be both unforgeable and verifiable by any other user. If the binding was forgeable it would be possible for an adversary to masquerade as the owner of the public key; similarly true for unverifiable bindings. In addition, it must be possible to revoke the binding.

We believe these four objectives represent a minimum set of criteria against which all global infrastructures should be evaluated. The DNS meets 3 of the 4 objectives, all but cryptographically verifiable bindings for its data. The proposed security enhancements add the functions and services necessary to meet the last objective. A global infrastructure based on the secure DNS is proposed that provides a solution to these objectives and can be used to distribute and manage the public keys of Internet users.

The Domain Name System (DNS) as deployed today is described first, followed by a description of the proposed secure DNS. The proposal is described last.

2. Domain Name System

The Domain Name System (DNS) is a distributed system for storing and retrieving resource records, a basic component that associates domain names with resources. Its most common use in the Internet today is to associate host names and IP addresses permitting the retrieval of one given knowledge of the other. Other uses include mapping old host names to new host names, identifying email gateways for hosts not directly connected to the Internet, and specifying other ancillary information necessary to the correct operation of the DNS.

The fundamentals below emphasize elements of the DNS that are relevant to the included proposal. An example is provided that can be easily contrasted with the example in the secure DNS section. This section ends with a discussion of trust issues, revocation, and how the DNS meets or does not meet the criteria stated above.

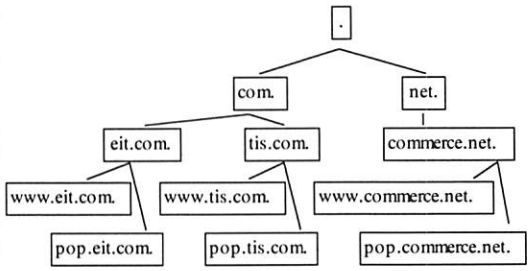
2.1 Fundamentals

Host names and IP addresses are two examples of resources of the DNS. Resources are accessed by a user or user application interacting with a resolver. The interaction includes the indication of a domain name and a resource associated with the name that is desired. The resolver interacts with one or more servers to obtain the requested resource. The interactions between a user and a resolver are a local implementation issue and will not be discussed further. The DNS specifications define the protocol used between the remaining elements.

Resource records are comprised of a domain name, a type field indicating what resource is contained within it, the data that is the resource, and other ancillary information. A domain name is comprised of an ordered sequence of labels which when displayed are usually separated by a dot (.). The type field implicitly indicates the syntax of the resource record and permits many kinds of resources to be associated with a domain name. The data is interpreted according to the type field. Some ancillary information will be considered later.

Domain names are chosen from a tree-structured name space. A domain name is either a leaf or an interior node of the tree space. Each leaf node holds a set of resource records. An interior node also holds a set of resource records, some of which will provide information about other nodes in the tree. Servers hold information about the tree structure and resource records.

The tree begins with a root designated by a single dot (.). Labels are added to the left, separated by dots, adding depth (a new level) to the tree and indicating nodes further away from the root. All labels at the same level in the same branch of the tree are required to be unique. Each node in the tree is named by concatenating the labels of the nodes in the tree along the path to the root. This is depicted below.



Some servers know that they hold all the resource records for a part of the tree; these servers are designated authoritative for that set of resource records. Although any server may respond to any query it receives for which it has the requested resource record, only authoritative servers may return authoritative resource records in response to a query.

Authoritative data is organized into zones. Zones are managed by servers. A primary server is always pre-configured with its zones. There is always at least one and may be several secondary servers that will automatically download the zone from the primary.

2.2 Example

By way of example, suppose a user application needs the IP address of the host "www.commerce.net". The user application would invoke a local resolver that accepts responsibility for obtaining the IP address.

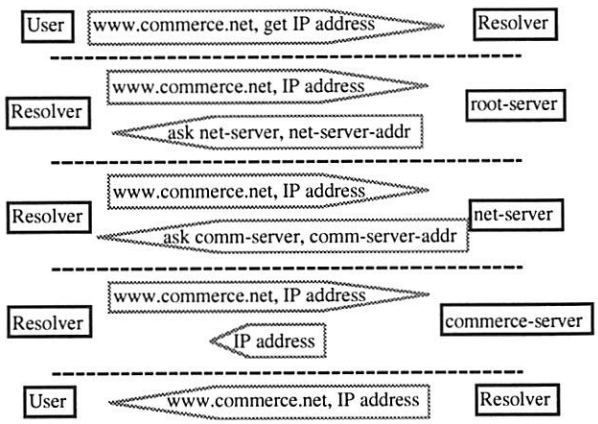
The local resolver would have been pre-configured with the IP address of the root-server. It would begin by asking the root-server for the IP address of "www.commerce.net". Since the root-server is authoritative for the root, it knows that it does not have an authoritative response for resources in the net-domain. It returns the hint telling the local resolver to ask net-server. It includes glue in the form of the IP address of net-server so that it can be contacted directly.

The local resolver then asks net-server for the IP address of "www.commerce.net". Since net-server is authoritative for net, it knows that it does not have an authoritative response for resources in the commerce-domain. It returns the hint telling the local resolver to

ask commerce-server. As was done by the root, it includes glue in the form of the IP address of commerce-server so that it can be contacted directly.

Finally, the local resolver asks commerce-server for the IP address of "www.commerce.net". Since commerce-server is authoritative for the commerce-domain it knows the IP address of domain names in its domain. It returns the requested IP address.

The basic algorithm is depicted below.[1]



2.3 Trust

The DNS functions today principally because all parties to the system cooperate and agree to function according to the specifications. In particular, this means that all servers respond truthfully to all queries and do not return any misleading information. As a result the DNS can be exploited in several ways [12,13], although the example above demonstrates one of particular interest that, in fact, is required by the DNS specification.

An authoritative server may delegate branches of the part of the tree it owns, however it is not authoritative for any of the data in the delegated branch. In particular, a server will know it does not have the answer to a query but it can only provide a hint as to where the correct answer may be found. The hint it provides is the name of one or more servers that are more likely to have the authoritative data requested. Interestingly, the complete set of authoritative servers for the requested data is only available from the server that has the authoritative data. Although the set in the authoritative server and its parent server are supposed to be equal, there is no enforcement mechanism and no existing implementation checks or cares.

[1] This example is for expository purposes only and does not represent a recommended implementation strategy.

The issue is compounded by the fact that knowing the name of a server more likely to have an answer is insufficient information to proceed. A resolver needs to know the address of the server in order to contact. However, if the server is within the domain to be contact, a typical configuration, the authoritative address exists only within the server itself and thus is unattainable. As a result, servers returning hints are required to provide the glue necessary to facilitate contact by the resolver, with which the servers must have been pre-configured.

The issue is the elements of the DNS architecture are not tightly-coupled. There is no explicit or enforced relationship between a branch in the tree and its parent.

2.4 Revocation

Consistent with the DNS trust model, a revocation mechanism exists that works if all participants cooperate and function according to the specifications. In principle, a resource record binds a domain name to the data contained within it. This binding is defined to be valid until the time-to-live (TTL) field specified within the resource record expires.

When an authoritative server returns data, it specifies a TTL value in the resource record. The value is the number of seconds the recipient of the data may cache the data and use it to respond to future queries it may receive. When the TTL expires the receiver is required to destroy its local copy and re-query the authoritative server for it.

An authoritative server does not expire the data for which it is authoritative. When responding to queries it always responds with the TTL set to its initial value. To remove data from an authoritative server it must be removed from its configuration files.

2.5 Criteria Evaluation

The DNS as deployed in the Internet today meets three of the previously stated criteria.

global availability - Sites on the Internet must have a DNS server on the Internet binding their host's names to their IP addresses in order to effectively use the services available to them. The DNS is an infrastructure protocol that most, if not all, sites connected to the Internet support. As a result, the DNS is both globally available and scaleable.

globally unique and unambiguous names - By definition, the DNS defines tree structured name spaces, each with a unique root, therefore each guaranteeing that a name within it is globally

unique and unambiguous. If DNS names can be mapped onto the names users prefer to associate with their public keys, the requirement for globally unique and unambiguous names would be met.

real-time access to public keys - The DNS provides real-time access to the resources it manages for the IP-connected Internet. Although the DNS is not currently used to distribute and manage public keys, it is independent of the resources that it manages. Therefore, by defining a resource record for storing public keys, it could be used for distributing and managing public keys.

However, the binding of names and objects in the DNS is based on cooperating peers. All peers are assumed to be honest and to respond to all queries with the correct answer. Thus, although the DNS is ideally suited to the needs of a public key distribution and management infrastructure, it does not meet the requirement of providing a cryptographically verifiable binding between names and objects.

3. Secure Domain Name System

Security enhancements for the DNS [9] have been drafted and submitted for consideration as a Proposed Standard in the Internet. The enhancements include the security services of data integrity and data origin authentication, noting that a digital signature mechanism could support both services. The objective of the enhancements is to cryptographically bind domain names to their resources, i.e., digitally sign the resource records managed by the DNS. Of course, backward compatibility with the existing, deployed DNS is supported.

The fundamentals below emphasize elements of the secure DNS specification that are relevant to the included proposal and assume familiarity with basic DNS terminology as defined in [5]. The example from the DNS section is enhanced to show how the addition of security affects the operation of the DNS. This section ends with a discussion of trust issues, revocation, and how the secure DNS meets or does not meet the criteria stated above.

3.1 Fundamentals

The specification defines two additional resource records that are relevant to this discussion: signature (SIG) and key (KEY).

The SIG resource record stores the signature calculated over a set of other resource records and other ancillary information, including the domain name of the signatory who created the signature and a footprint of which of potentially many signatory keys was used to create the signature.

The KEY resource record is used for storing public keys, which initially will be used by the secure DNS itself to distribute and manage the public keys it needs in support of its security services. The ancillary information stored with the public key includes an indication of the purpose for which it may be used, e.g., in support of a specific application.

A zone supported by a DNS server that creates, verifies, distributes, and manages SIG and KEY resource records is called a secure zone. All other zones are insecure. There is ancillary information in the KEY resource record that permits a server to authoritatively and securely indicate that a zone is insecure.

In a secure zone, a SIG resource record is created for each type of resource record in each domain. In addition, a SIG resource record is created for all the resource records in a domain, to be used when responding to an ANY query so that a resolver can check that all records are present, and a SIG resource record is created for all the resource records in a zone, to be used to validate zone transfers.

The authority for the public key corresponding to the private key for a secure zone is the super zone, i.e., a secure zone is not authoritative for its own key, although it may be authoritative for the public keys of domains within it.

All the security enhancements to the DNS are algorithm independent. However, to enhance interoperability among deployed systems, the specification has chosen an initial set of algorithms that must be supported by all implementations: RSA and MD5.

The proposed security enhancements will function automatically if the public key(s) of the root zone is ubiquitously known and manually configured. The specification also allows servers to decide locally which sub-trees they believe are security conscientious as opposed to believing the entire name space.

An overview of each of the basic operations of secure DNS is described below.

3.1.1 Secure DNS Signature Creation

The basic signature creation operation is as follows.

All the resource records within a domain name within a secure zone are grouped according to their type, e.g., all the IP addresses of a given host are grouped together.

Each group of resource records is canonicalized. The canonicalization involves at least expanding any compressed names in the resource record and sorting the records. The objective of the canonicalization step is to ensure a unique and unambiguous representation of the data to be signed.

The MD5 hash of each group of canonicalized records is computed.

The private key of the secure zone in which the domain name appears is used to create a digital signature for each group by signing its computed hash value. This signature is stored in a SIG record for the type over which it applies, along with the name of the signatory.

Conceptually, if the resource record being signed is a KEY record, the KEY and SIG record combination represent a certificate that cryptographically binds the domain name of the KEY resource record to its public key. Retrieval of the certificate requires retrieving both the KEY resource record and its corresponding SIG resource record

3.1.2 Secure DNS Signature Validation

The basic signature validation operation is a two step process. First the resources records of interest are verified as follows.

All the resource records of a given type for a given domain name are retrieved, along with the relevant signature record. Normal operation of the DNS would always return all records of a given type when queried for a type. Normal operation of a secure DNS server will also return the signature resource record.

The public key of the signatory indicated in the signature (SIG) record is retrieved and used to verify the signature of the resource records.

The resource record is considered valid if the signature can be verified using a valid public key.

Second, the public key of the signatory used in the second step above must itself be validated. The KEY resource record is a distinct record type and therefore has its own corresponding signature record. Consequently, validating the public key of the signatory is accomplished by invoking the signature validation process on the KEY resource record in which it is stored. This process is repeated recursively until the public key of a trusted point is used to verify a signature, as described in Sections 3.2 and 4.2.

A detailed discussion of various implementation optimizations and issues associated with validating signatures can be found in [10].

3.1.3 Secure DNS Server Operation

The behavior of security aware servers is enhanced as follows.

When responding to a query for data in a secure zone, by default only data for which the signature has been verified by the server is returned. Security aware clients may request that unverified data be included in a response, which servers must honor.

When responding to a query for data in a secure zone, both the resource record and its corresponding signature record must be returned.

The default behavior requirement that security aware servers may only respond with data for which they have verified the signature supports a very important service for non-security aware clients. It is likely that there are many (orders of magnitude) more DNS clients than there are DNS servers and that the transition to security aware servers will progress more quickly than the transition to security aware clients. If this is true, non-security aware clients will be used for quite some time after the initial deployment of security aware servers. Having security aware servers return only records for which they have verified the signature allows local[2] non-security aware clients to take advantage of the security services without any changes.

3.2 Example

The same example from the previous section is expanded here to include the additional security functionality.[3] To simplify identification of the signatory, in this example the secure zone is served by a server named within the zone itself.

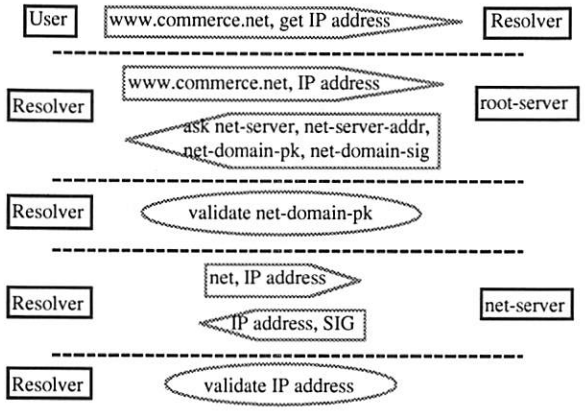
Recall, suppose a user application needs the IP address of the host "www.commerce.net". The user application would invoke a local resolver that accepts responsibility for obtaining the IP address.

The local resolver would have been pre-configured with the IP address and public key of the root-server. It would begin by asking the root-server for the IP address of "www.commerce.net". Since the root-server is authoritative for the root (.), it knows that it does not have an authoritative response for resources in the net-domain. It returns the hint telling the local resolver to ask net-server. It includes glue in the form of the IP address of net-server so that it can be contacted

[2] Use of the word local is intended to remind the reader that the client may be vulnerable over its link to the server. Presumably, a client believes its local environment is not vulnerable to adversaries and therefore it can believe responses received from local servers without verifying them.

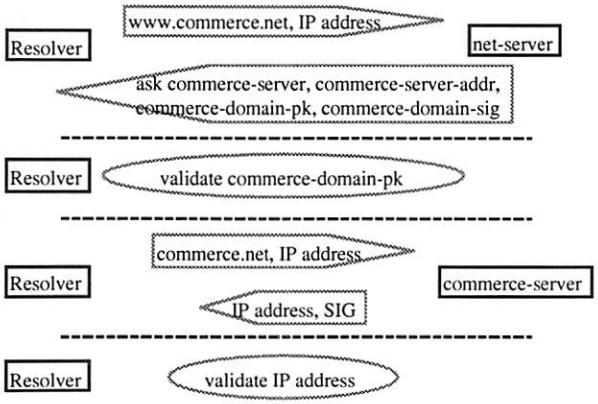
[3] As before, this example is for expository purposes only and does not represent a recommended implementation strategy.

directly. In addition, the public key for net-domain and its corresponding signature resource record is included in the response.



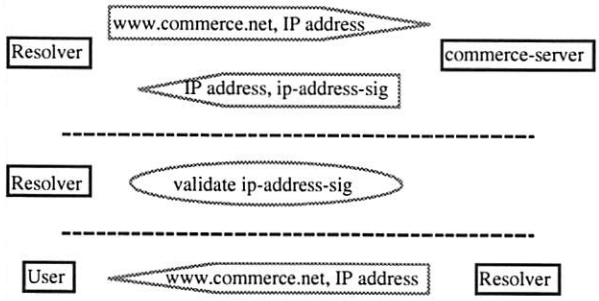
The local resolver must first validate the public key for net-domain using its pre-configured copy of the root's public key. If the public key is valid it must then proceed to validate the glue provided by the root-server. To do this it must first ask net-server for its address records. Net-server will return the complete set of address records for itself and the corresponding signature record. The local resolver must use its copy of net-domain's public key to validate the signature on the address records.

The local resolver then asks net-server for the IP address of "www.commerce.net". Since net-server is authoritative for net-domain, it knows that it does not have an authoritative response for resources in commerce-domain. It returns the hint telling the local resolver to ask commerce-server. As was done by the root, it includes glue in the form of the IP address of commerce-server so that it can be contacted directly, as well as the public key of commerce-domain and its corresponding signature record.



Again, the local resolver must first validate the public key for commerce-domain using its copy of net-domain's public key. If the public key is valid it must then proceed to validate the glue provided by net-server. To do this it must first ask commerce-server for its address records. Commerce-server will return the complete set of address records for itself and the corresponding signature record. The local resolver must use its copy of commerce-domain's public key to validate the signature on the address records.

Finally, the local resolver asks commerce-server for the IP address of "www.commerce.net". Since commerce-server is authoritative for the commerce-domain it knows the IP address of domain names in its domain. It returns the requested IP address and its corresponding signature record. The local resolver must validate this one last signature by using its copy of commerce-domain's public key.



3.3 Trust

Whereas the currently deployed DNS functions principally because all parties agree to be truthful at all times, when following authoritative data with secure DNS it is only possible to lie about the data for which a server is authoritative. Therefore, although a server can mislead clients attempting to contact it or any of its sub-zones, it is not possible to mislead a client about any other zone.

The DNS manages tree structured, hierarchical databases with single, globally unique roots. Just as the delegation of authority in the DNS passes down the tree from the root, so does the trust in the secure DNS. In the secure DNS, the trust begins at the root zone and is passed down to each sub-zone when its parent signs the sub-zone's KEY resource record. The delegation absolves the parent of all responsibility for the sub-zone except to indicate its existence and where to find it. The digital signature provides undeniable proof of the

parent's intent to delegate, although it says nothing about the sub-zone's acceptance of the delegation.

A feature of having a secure zone be authoritative for the KEY records of its sub-zones is that the secure DNS architecture is tightly-coupled. From the root in the secure DNS, it is always possible to move securely to any other zone. Of course, a zone could respond with false information about itself and its sub-zones, but doing so only prevents legitimate access to itself. However, the trust relationship is not the same when moving up the DNS tree.

According to the secure DNS specification, a secure zone is required to have a KEY record with the public key of its superzone in its zone signed by itself. The purpose of this record is to make it possible to securely move up the DNS tree. This permits sites to pre-configure the public keys of arbitrary secure zones instead of the root, presumably on servers "close" to them, and still be able to move securely to any other zone.

Unfortunately, authority in the DNS, in particular the delegation of zones, passes down the tree not up the tree. Although it may be possible to trust the "close" servers of the pre-configured zones to be truthful about the public key of their parent zones, prudence suggests that the transitivity of that trust be limited. Whereas moving down the tree limits the damage a malicious server can inflict to data for which it is authoritative, a server that lies about its parent could mislead a client about a far greater portion of the tree. In the extreme, it could masquerade as a root to clients moving up the tree and subvert the entire name space.

Another feature of being able to pre-configure the public keys of arbitrary secure zones is to permit full use of the security features between disjoint fragments of the tree. Until secure DNS supplants the currently deployed DNS, there will be portions of the tree that implement security but whose parents do not. These portions of the tree could advertise their public keys, via several different mechanisms to enhance validation, thus permitting other secure portions of the tree to pre-configure the public keys and be guaranteed of obtaining correct information.

3.4 Revocation

The secure DNS specification does not include an explicit mechanism for revoking the bindings created by digitally signing resource records. Instead the DNS revocation mechanism was enhanced by having the digital signature of a set of resource records cover the initial value of the TTL field. Only the initial value is protected because protecting the values decremented by servers who have cached the resource record for a while would require those servers to keep a private key on-line at all times to be able to sign resource records in real-time.

The normal operation of the DNS requires clients to re-query for data for which the TTL field has expired. Protecting the initial TTL value guarantees that a client will re-query for data at least that frequently. This is functionally equivalent to an explicit revocation mechanism.

When using an explicit revocation mechanism, validation of a binding requires that a list of revoked bindings be retrieved and inspected to see if the binding of interest is listed on it. The revocation list is updated on a regular basis, at a minimum frequency usually indicated in the list and commensurate with the perceived significance of the binding. To leverage the DNS TTL field as a revocation mechanism, its value should be set to the minimum frequency an explicit revocation list would be updated if it were used.

During normal operation, a server would check to make sure the TTL on a resource record had not expired prior to using it in a response. Checking the TTL is analogous to checking for membership on an explicit revocation list.

In order for a secure zone to revoke a binding it either removes the invalid resource records from its configuration or it replaces them with valid records. All other servers will acquire the new bindings no later than the value of the TTL from time the valid records are inserted.

Since the security of the entire system is dependent on the use and management of valid public keys, the key bindings are more significant than other bindings. As a result, the TTL for KEY resource records should probably be substantially less than that used for other records, e.g., if 30 days is used for most records, 7 days would be a good choice for KEY records.

It is important to remember that the expiration of the signature is distinct from the TTL. The secure DNS specification includes a maximum lifetime for each signed resource record. During normal operation a signature could be valid for 1 or more years in contrast to the 1 or more months of the TTL.

3.5 Criteria Evaluation

Since secure DNS enhances or adds to the existing DNS, as opposed to changing or removing from the existing DNS, it inherits the meeting of the same criteria as the existing DNS. In particular, global

availability, globally unique and unambiguous names, and real-time access to public keys.

One potential issue with respect to global availability is whether or not secure DNS will scale for global use. The DNS database (actually each of several) is predicated on the existence of a single global root domain. On the one hand, PEM may epitomize the likelihood that the Internet community will embrace an infrastructure structured in this way, i.e., it will fail.

On the other hand, there are at least two, more positive outcomes. The first is that the DNS has been functionally exceptionally well with a single global root. The space is managed by the Internet Assigned Numbers Authority and, setting aside the various trademark, copyright, and other related legal issues, there is acceptance and support for the process. It is entirely possible that secure DNS may succeed with a single global root in spite of the failure of PEM.

The second possibility is that secure DNS could be deployed with multiple "roots". Instead of a single root domain, setup each top-level domain name to be the root of its hierarchy. This would require the maintenance of multiple root public keys, which would require the establishment of a distribution mechanism. However, given that the DNS handles the problem of multiple servers for the root domain, it is likely that a similar kind of solution will suffice for the management of multiple root keys.

In any case, this issue will be resolved as soon as we begin to see global usage of secure DNS. As of this writing, at least one reference implementation is being developed and it is currently in beta test. In fact, there are two significant implementation issues that are relevant to the scaleability of the secure DNS.

First, the addition of public keys and their signatures to the DNS will stress the robustness and reliability of current DNS implementations. For example, the most popular DNS implementation maintains its entire database cache in memory. For 1024-bit RSA keys, 500 user keys will require at least 1-megabits of memory just to store the public keys and their signatures in the cache, which does not include the memory needed to store the 500 domain names and header information of each of the key and signature records. While it is true that hardware and memory are getting cheaper all the time, will it be cheap enough that the "com" domain, for example, can be managed by one server?

Second, as is always the case with public key cryptography, the protection of the private key is the cornerstone of the security provided by the entire system. The secure DNS specification recommends that the signing of resource records (the creation of the SIG resource records) be completed off-line, and that the database file with the original resource records and their signatures be transferred manually to the on-line primary server. Although this process may appear cumbersome, in principle, it should not be a frequent occurrence. Alternatively, other technologies could be explored for protecting the private key of the zone, e.g., a trusted system, which provides guaranteed access control, and PCMCIA cards, which keep the private key in a physically secure location.

Finally, secure DNS meets the last criteria that was not met by the existing DNS: cryptographically verifiable bindings between names and public keys. By definition, secure DNS cryptographically binds domain names to the remaining data in the resource record. Since there is a resource record expressly for the purpose of managing public keys, the secure DNS meets this requirement.

4. User Public Keys and Secure DNS

The secure DNS uses public key cryptography in support of its security services. It provides for itself a global key distribution and management system, which is exactly what users and user applications need to support their security services.

If each user was assigned a domain name, then each users' public key could be stored as a KEY resource record in their domain. From the point of view of the secure DNS, retrieving and verifying these KEY resource records is no different than retrieving and verifying the KEY resource records it uses during its normal operation. User applications could be modified to query the secure DNS with the user's domain name whenever the user's public key is needed.

4.1 User Domain Names

In the Internet, we already have a well-understood, globally deployed, tree-structured name infrastructure for users: RFC822 [11] email addresses. Many users are already commonly known by their email addresses. Internet-connected users include their email address on their business cards. Many companies are known by their domain name and some provide a variety of well-known email addresses through which they may be contacted.

To use email addresses there must exist rules for mapping them to DNS names. For a large fraction of Internet users this will be a straightforward process. RFC822 email address all have the syntax "local-

part@domain-part". By definition, domain-part is a domain name so it is easily appended to the result of mapping local-part to a legal sequence of one or more domain name labels.

In the simplest case, the local-part conforms to the syntax of a domain name label, i.e., it contains only the letters A-Z and a-z, the digits 0-9, and the hyphen, e.g., galvin@commerce.net. The at-sign (@) character could be replaced with a dot (.) resulting in an email address that becomes the user's domain name.

It is incrementally more complicated to permit dots and other punctuation characters to appear in the local-part. Other punctuation marks could be replaced with dots, collapsing multiple adjacent dots to a single dot. This will require the creation of additional branches in the DNS tree, but that is a one time administrative exercise for each user, easily completed as part of creating a new user account.

It is incrementally more complicated again to provide mapping rules for other printable yet illegal characters, e.g., parenthesis and quotation marks. An approach that works is to replace illegal characters with strings of legal characters, similar to the approach taken in [14].

This simple set of rules will serve to provide automatic processing for a large fraction of the email user community. For more complicated local-parts, for example X.400 addresses, more complex mapping rules would have to be developed.

4.2 Validation Issues

Believing cryptographically verifiable bindings requires that a trusted path exist from the binding of interest to a trusted point. By default, the most trusted point in the secure DNS is the root. Conceptually, the root signs the public key for each of its delegated zones, and makes each key and its respective signature resource record available via the DNS. Similarly, delegations at other points in the tree would have their key records signed by their super zone at the point of delegation.

Validating a signed resource record constructs a sequence of KEY resource records, the first of which contains the public key of a trusted point. This public key is known *a priori* to be valid and trusted. It is used to verify the signature in the SIG record of the next KEY record in the sequence. Similarly, this next key is used to verify the signature in the SIG record of the next KEY record in the sequence, until the sequence terminates with the KEY record containing the public key needed to verify the signature of the signed resource record. Consider the example detailed in Section 3.2.

Since the secure DNS is a homogeneous system, all of the public keys created exist for a single reason and are implicitly signed according to a uniform policy. However, when keys for other purposes are added to the DNS, there is no implicit policy. In the absence of an explicit policy, it is not possible for an application to automatically evaluate whether a retrieved key is suitable for its intended use, irrespective of the existence of a trusted path. The only evidence a trusted path provides is that the name associated with public key is correct.

As a result, the DNS provides a mechanism suitable for assisting in the management of globally deployed public keys, but it is insufficient to support in and of itself the knowledge of whether a public key is suitable for a particular use. Additional ancillary information is necessary, e.g., a policy that defines the valid uses of the public key.

However, where human interaction is possible and likely, e.g., with secure email applications, a user could evaluate in real-time whether or not a public key was appropriate for use. In such an application, the secure DNS is a necessary and ideal enabling technology to the global deployment of the application.

5. Conclusions

The continued standardization of the use of public key technology in the Internet demands a globally available public key distribution and management system. The Domain Name System (DNS) is an infrastructure protocol which provides an ideal base on which to build such a system. The secure DNS specification being designed and implemented includes a public key distribution and management system that could be used to manage users' public keys if users were assigned domain names.

With a domain name users could store their own public keys as resource records where they would be quickly and easily accessible by others. This is straightforward to do for the vast majority of Internet users by taking their email addresses and changing the at-sign (@) to a dot (.). Additional suggestions were proposed for incrementally more complicated email addresses.

The use of the secure DNS as a public key distribution and management system for users will require changes to application programs. However, this transition can not proceed until there exists a reference implementation. As of this writing, one is being developed and is currently in beta test. However, as indicated above, the scaleability of existing implementations is uncertain.

Finally, using the DNS to validate public keys for users begs the question of under which policy was the public keys are signed. This is an issue being addressed in the context of X.509 certificates by many different organizations, but no attention to date has been given to resolving the issue in the secure DNS.

6. References

[1] John Linn. Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures. RFC1421, February 1993. Obsoletes RFC1113.

[2] Steve Kent. Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management. RFC1422, BBN Communications February 1993. Obsoletes RFC1114.

[3] David M. Balenson. Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers. RFC1423, Trusted Information Systems, February 1993. Obsoletes RFC1115.

[4] Burton S. Kaliski. Privacy Enhancement for Internet Electronic Mail: Part IV: Key Certification and Related Services. RFC1424, RSA Laboratories, February 1993.

[5] Paul Mockapetris. Domain Names - Concepts and Facilities. RFC1034, ISI, November 1987. Obsoletes RFC973.

[6] Paul Mockapetris. Domain Names - Implementation and Specification. RFC1035, ISI, November 1987. Obsoletes RFC973.

[7] Paul Mockapetris. DNS Encoding of Network Names and Other Types. RFC1101, ISI, April 1989. Updates RFCs 1034, 1035.

[8] Bill Manning and Richard Colella. DNS NSAP Resource Records. RFC1706, ISI and NIST, October 1994. Obsoletes RFC1637.

[9] Donald E. Eastlake and Charles W. Kaufman. Domain Name System Security Extensions. Work in Progress.

[10] James M. Galvin and Sandra L. Murphy. Using Public Key Technology - Issues of Binding and Protection. INET'95, Internet Society, June 27-30, 1995.

[11] David H. Crocker. Standard for the Format of ARPA Internet Text Messages. RFC822, University of Delaware, August 1982.

[12] Steven M. Bellovin. Using the Domain Name System for System Break-ins. The Fifth USENIX UNIX Security Symposium, Salt Lake City, June 5-7, 1995.

[13] Paul Vixie. DNS and BIND Security Issues. The Fifth USENIX UNIX Security Symposium, Salt Lake City, June 5-7, 1995.

[14] S. Hardcastle-Kille. Mapping between X.400(1988) / ISO 10021 and RFC 822. RFC1327, ISODE Consortium, May 1992. Obsoletes RFC987, RFC1026, RFC1138, RFC1148. Updates RFC822.

Compliance Defects in Public-Key Cryptography

Don Davis*

May 29, 1996

Abstract

Public-key cryptography has low infrastructural overhead because public-key users bear a substantial but hidden administrative burden. A public-key security system trusts its users to validate each others' public keys rigorously and to manage their own private keys securely. Both tasks are hard to do well, but public-key security systems lack a centralized infrastructure for enforcing users' discipline. A *compliance defect* in a cryptosystem is such a rule of operation that is both difficult to follow and unenforceable. This paper presents five compliance defects that are inherent in public-key cryptography; these defects make public-key cryptography more suitable for server-to-server security than for desktop applications.

1 Introduction

Public-key cryptography is uniquely well-suited to certain parts of a secure global network. It is widely accepted that public-key security systems are easier to administer, more secure, less trustful, and have better geographical reach, than symmetric-key security systems. However, it is *not* widely appreciated that these advantages rely excessively on end-users' security discipline. In fact, the reason public-key security doesn't need a trusted key-management infrastructure is that the burden of key-management falls to public-key clients. With public-key cryptography, clients must constantly be careful to validate rigorously every public key they use, and they must husband the secrecy of their long-lived private keys. It turns out that these tasks are harder than they seem.

End-users are unwilling or unable to manage keys diligently. Perhaps surprisingly, it's impossible to automate asymmetric key management completely; certain security details remain for human intervention, such as Root-key validation, passphrase choice, and *clients'* physical security. Even where automation is possible, as with revocation-list checks, scaling problems and performance costs make short-cuts likely. If users or developers skip these details, there is no way to detect their omission or to audit the consequences. I have coined the name *compliance defect* for this situation: a rule of operation that is difficult to follow and that cannot be enforced. Compliance defects undermine the security of public-key cryptography. When users fail to manage their private keys securely, or when they fail to validate each other's public keys rigorously, then authenticity and privacy guarantees weaken, and everyone's security deteriorates. Users' behavior is the weak link in any security system, but public-key security is unable to reinforce this weakness.

This is not to say that compliance defects make public-key systems worthless. Rather, compliance defects just make public-key security unsuitable for desktop applications. Only sophisticated users, like system administrators, can realistically be expected to meet fully the demands of public-key cryptography. Accordingly, I suggest that public-key cryptography is best suited to securing communications between servers, between sites, and between organizations. Only in such large-scale infrastructural applications does public-key's geographic reach justify its substantial administrative burden of constant vigilance.

2 Public-Key Infrastructure: Review

A public-key security system comprises three infrastructural services.

- The *Certification Authority* (CA) signs users' public keys,
- The *Directory* is a public-access database of valid certificates,
- The *Certificate Revocation List* (CRL) is a public-access database of invalid certificates.

*Affiliation: Independent Consultant, 1318 Comm. Ave #16 Allston, MA 02134; *don@mit.edu*

In a wide-area network, each of these services may be deployed as a hierarchy of servers. For example, in a hierarchy of CAs, each CA has its own public-key, which is signed into the CA's own certificate by the next higher CA in the tree. However, the "Root CA," at the top of the hierarchy, has no one to sign its public key.

In the rest of this section, I summarize the administrative features of public-key security systems, with emphasis on the issues of key-handling discipline that I'll describe in the rest of the paper. [1] Key-management issues arise at the following crucial moments in the life-cycle of a user's public key certificate:

1. Key-Creation:

- The user creates a new key-pair.
- The user proves his identity to the CA (not electronically).
- The CA signs a certificate that names the user as the bearer of his new public key.
- The user also receives the Root CA's public-key, for later use.
- The user chooses a secret passphrase, and uses it to encrypt his asymmetric private key.

2. Single-Sign-On

- At login, the user types his passphrase, so as to decrypt his private key.
- With his private key, the user participates in public-key protocols.

3. Authenticating Others

- To communicate securely with other users and with networked services, the user refers to the other parties' public-key certificates.
- The user either exchanges certificates directly with other users, or he get others' certificates from the Directory service.
- Before using a certificate, the user must check the CRL for notice of the certificate's revocation, and must
- Validate the CA's signature. This step is recursive, and ends with the *out-of-band* validation of the Root CA's public key.

4. Password-Change

- The user should regularly change the passphrase with which he decrypts his asymmetric private key.

5. Key-Revocation

- Certificates are timestamped to expire after a few months or a year.
- If a user's passphrase or his private key is compromised, then he must inform the CRL administrator, who disseminates a notice that the corresponding public-key certificate has been revoked
- The user should check the CRL every time he uses a certificate, because the CRL may be updated at any moment.

The reader will notice, on reviewing this chronology, that a public-key user is frequently required to protect and validate a variety of symmetric and asymmetric keys. Unfortunately, a public-key infrastructure cannot help the user in these tasks, nor can it compel his compliance.

3 Compliance Defects

Public-key Cryptography has five unrealistic rules of use, which I call *compliance defects*. These defects correspond one-for-one with the crucial moments in a key-pair's life-cycle:

1. Authenticating the User (Issuance): How does a CA authenticate a distant user, when issuing an initial certificate?
2. Authenticating the CA (Validation): Public-key cryptography cannot secure the distribution and validation of the Root CA's public key.
3. Certificate Revocation Lists (Revocation): Timely and secure revocation presents terrific scaling and performance problems. As a result, public-key deployment is proceeding *without* a revocation infrastructure.
4. Private-Key Management (SSO): The user must keep his long-lived private key in memory throughout his login-session.
5. Passphrase Quality (PW-Change): There's no way to force a public-key user to choose a good passphrase.

[1] A similar summary for symmetric-key security systems appears in the Appendix. My outlines are loosely based on the X.509 [11] and Kerberos systems [15, 20].

The creation and revocation defects can in principle be remedied with centralized infrastructure, but the other three problems cannot be solved without the user's cooperation and intervention. Further, extra infrastructure entails scaling problems that lead immediately to shortcuts, trading away security in favor of performance. Recent proposals for public-key-based commerce have typically been naive in just this way, by ignoring the need for this extra infrastructure when in reality, the need can but momentarily be deferred.

3.1 Authenticating the User

One of the great promises of public-key cryptography is that a Certificate Authority can serve many more users than can a key-distribution service, because users only rarely have to interact with the CA. Indeed, since users get certificates monthly or annually, a CA's per-user load is a hundredth to a thousandth of a key-server's. Even accounting for the CA's greater crypto overhead, we might expect a CA to serve up to a million or more users. Unfortunately, this cheap scaling is a false promise.

The fly in the ointment is that there's more to issuing a certificate than merely calculating a digital signature. A public-key certificate is an assurance about the identity of the corresponding private key's owner. Just as the user can't trust an electronic delivery of the Root-key (see below), the CA can't trust electronic assurances of new users' identities. In both cases, what's required is a secure "out-of-band" communication. Ideally, the CA sys-admin should personally meet the new user and check his identifying documents (driver's license or passport), before signing a statement about who holds what key. Such face-to-face meetings are routine for user-account creation in the smaller installations that use symmetric keys, but truth be told, meeting a million users face-to-face isn't easy. Further, in the public-key world, users and CAs usually are widely separated, so universal face-to-face certificate issuance is really practical only for PGP hobbyists [25]. Properly authenticated certificates will have to be expensive, because of the labor cost in a face-to-face identity check. For most people, a certificate's ownership will be no more airtight than a credit card's privacy, especially since credit-card companies will be issuing certificates. Often it will be even weaker than this, since Verisign is planning to issue its lowest-level certificates by e-mail. Perhaps the only U.S. organization that already has the infrastructure necessary for correctly issuing public-key certificates is the U.S. Postal Service [21]. Certainly, the degree of certainty that one requires for a certificate varies with the application. A full discussion of such issues and their relevance to the present argument is beyond the scope of this paper, but it's clear that unauthenticated issuance offers *no* security guarantees.

It is possible to use a symmetric-key security system to authenticate a public-key certification-request. MIT has added a PGP-signing service to the Kerberos authentication system. [18] In this scheme, the authenticity of the certificate's name-to-key binding is as sound as the Kerberos account's creation was. If the user-accounts administrator checked IDs in face-to-face meetings, the Kerberized CA's certificates will be meaningful. If instead the users can register themselves remotely, then the certificates will be all but meaningless. [2]

In sum, CAs cannot scale as well as might seem possible, because account-initiation is less a technical scaling problem than a social one.

3.2 Authenticating the CA

It is a telling and ominous fact that every electronic-commerce protocol specification explicitly disavows all responsibility for the validation of the Root CA's public key [14, 16, 22, 23]. "Outside the scope of this document" is a typical waiver. [14]

Before using a public-key certificate, a user must authenticate it by checking its certifying signature and the signature on each public key in its chain of certifying authorities. It's commonly forgotten that public-key cryptography cannot afford the user any automatic procedure for validating the top-level CA key. To make sure the top-level CA key is authentic, the user has three choices:

1. Hand-checking it against an authentic paper copy;
2. Making sure that the CPU's copy is incorruptible;
3. Using a separate security system, like a smartcard or Kerberos [15, 20], to convey an authentic copy to the CPU.

It is not sufficient to pass the top-level keys inside the application-client's executable, as Netscape's Web browser does [14, 5]. Even if the executable is signed, we still have to authenticate the signature's validation-key. [3] If the attacker replaces one of the client software's top-level CA keys, by patching the

[2] At MIT, staff members usually get their accounts face-to-face, but students usually do it remotely.

[3] It is often suggested that the Root-CA can sign the executable that contains the Root-CA key, but an attacker can do this with a false CA-key, too. Such self-signed certificates are essentially meaningless.

executable, for example, then the attacker can cause the client to accept forged certificates. Unhappily, when executables come as freeware and from file-servers, this key-substitution is easy to do. Once the user accepts a forged certificate, the attacker can pose as the application server. To escape detection, the attacker just plays as a man-in-the-middle. To prevent the MITM properly, a public-key protocol should sign the plaintext, then encrypt the signature, then sign the ciphertext – clearly a burdensome process. [4] It may be more practical to ensure that the Root key, once validated, cannot be corrupted. One way to guarantee this is to keep the Root key on a smart-card. Other, weaker safeguards are to keep the Root key under the passphrase's encryption, or equivalently, for the user to sign his copy of the Root-key himself.

3.3 Certificate Revocation Lists

Before we use a public key, we must validate the key's certificate in two ways: we must check the issuing CA's signature, and we must check the current Certificate Revocation List to see if the public key is still active. The CRL is part of the public-key infrastructure's account-management system. The other parts are the CA and its practice of issuing limited-lifetime certificates. It might seem that a user needs to check his cached certificates only when he acquires them, but this is not true. It is a simple matter for a virus to corrupt a certificate cache, and a certificate may be revoked just before use, or even just after it enters the cache.

A fair, if simplistic, rule of thumb is that the cost of key-issuance plus the cost of revocation is a constant [7]. For symmetric-key systems, these costs are roughly equal, but for public-key systems, certificate revocation is much harder than issuance. Revocation is the classic Achilles' Heel of public-key cryptography. When a user's public key must be removed from use, the only way to enforce prompt revocation is to check every certificate before use against a Certificate Revocation List. Naturally, CRLs must come from a secure and highly-available service. Further, to check the CRL server's own public-key certificate, the user must refer to a higher-level CRL server, because a CRL server cannot testify about its own certificate's currency. Thus, the public-key infrastructure needs a hierarchy of CRLs, just as it needs a hierarchy of CAs. However, this real-time reliance on a centralized infrastructure negates one of the main advantages of public-key cryptography – as originally conceived, public-key protocols would avoid dependence on centralized points-of-failure. Further, note that a rigorous check of a certificate's validity requires that the public key of each CRL in the chain to the Root has to be revocation-checked, just as with signature validations. However, while a signature-check takes around 10 milliseconds, a long-haul CRL check will often take 100 or more milliseconds. This extra performance burden makes it likely that applications will often avoid revocation-checking the CRL's own certificate. This compliance defect undermines the security of any application that uses public-key cryptography.

Clearly, the timely management of CRLs is an important scaling bottleneck. The size of the CRL can be minimized by using an access-control system to revoke access, but efficient ACL management for very large networks is another unsolved scaling problem. The need for prompt revocations becomes especially acute if digital signatures have financial or contractual import. There seems to be little progress towards a national CRL mechanism.

3.4 Private-Key Management

In order to use his public key for sending and receiving secure messages, a user must either enter his passphrase anew for each use, or he must keep his private key in memory throughout his login session. Note that clients apply the private key not only to sign and decrypt e-mail, but also to initiate sessions with secure network servers. Clearly, users will not accept software that forces them to re-enter passphrases all day long, so in practice their keys will stay in memory. This is a substantial security exposure, because it exposes a long-lived secret, the private key, to physical theft. For example, if the user leaves his keyboard unattended, or if his laptop is stolen, his private key will likely be compromised. Similarly, viruses and Trojan-Horse programs have been built to steal long-lived keys. [24] So, the private key is only as secure as the user's computer. Even if the user's private key is stored in an encrypting smartcard, so that the key itself doesn't reside in memory, the card still must stay in the reader throughout the session, and so is still vulnerable to theft.

Symmetric-key systems often are designed to replace long-lived passwords in RAM with short-lived session keys, so as to minimize the passwords' in-memory exposure Short-lived asymmetric key-pairs are not very useful, though. For signature operations only, it is possible to avoid keeping the private key on hand, by using a secret signature as a temporary private key. [8, 12] This temporary key's signature can

[4] Simply signing the ciphertext doesn't work; see [3, 10, 2].

be checked with the user's public key. [5] However, if the user is to be able to decrypt private messages, he has to keep his private key in memory, and in plaintext form, throughout his login-session. The only way to keep the in-memory key safe is to keep the user's computer physically secure, and to forbid all remote access by outsiders. Clearly, users will not reliably observe these precautions.

3.5 Passphrase Quality

A public-key system has no way to enforce expiration or quality controls on passphrases, because users don't share their passphrases with any security service or administrator. It's possible, of course, for the user's local passphrase-handling software to apply such controls, but if the user finds the controls inconvenient, he can just use a more lenient program to encrypt his private key.

Without effective passphrase-QA, users' private keys are only as secure as the filesystem on which they are stored. For example, if the encrypted private keys are stored on a networked filesystem, many will be utterly vulnerable to guessing attack. This threat makes it unsafe for a user to access the net from different machines, because such logins would require unauthenticated access to the user's passphrase-encrypted key, and so would expose the key to off-line dictionary attack.

In contrast, it's easy for a trusted-party KDC to enforce a quality-control policy on each user's password, because the user must share his password with the KDC, anyway. Typically, the KDC enforces the password-expiration controls when the user logs in, and only enforces the quality-controls when the user changes his password. Then, the KDC can apply various rules and filters to ensure that the password is hard to guess. For example, the Kerberos system has incorporated most of the features of U. Texas' `npasswd` command [9], password-expiration, and other password-QA features, in a flexible password-policy mechanism. [6] Another valuable approach is proactive dictionary-checking. Purdue's OPUS project developed a filter which quickly checks passwords against a set of dictionaries, but without divulging the passwords to any trusted party. [19]

Left to their own choices, users tend to choose passwords that are easy to guess, and they tend not to change their passwords unless the security system obliges them to do so. Thus, lacking effective password-quality controls, most public-key systems are vulnerable to off-line guessing attacks. An organization's first line of defense in data security is to enforce good password hygiene, so for corporate networks, this defect is a grave one.

Table 1 summarizes the compliance defects I have discussed.

4 Transferring Administrative Burdens

A symmetric-key KDC must be highly trustworthy and highly available. In comparison, a public-key system is easier to administer centrally, because a public-key infrastructure's trust and availability requirements are more relaxed:

- The CA doesn't have to be highly available, because users rarely need new certificates. The CA is a trusted service; it cannot eavesdrop on encrypted messages, but a corrupt CA can forge a key pair and certify it in a user's name. Thus, users do have to trust the CA not to issue false certificates.
- The Directory is in essence a convenience; it saves users the trouble of exchanging their certificates with each other. The Directory is unable to forge certificates, so it requires no trust, but it should be highly available.
- The CRL has to be trusted to disseminate revocations "promptly;" depending on the application's criterion for "promptness," this may require high availability.

So, we see that some components have to be trustworthy, and some have to be highly available, but trust and high-availability are generally not required simultaneously of each public-key service.

Symmetric-key systems have another administrative burden from which a public-key infrastructure is free. The public-key infrastructure is not a bottleneck in the network, because the CA, Directory, and CRL servers don't have to mediate in every secure communication, as a KDC must do. [7] This improves not only the network's performance, but its reliability,

[5] Let N be a public message, and denote its secret signature by N^d. To sign a message m, the signer calculates $S_m = (N^d)^m$; S_m and N together make up the temporary key's signature. To check the temporary-key signature, the verifier calculates ${S_m}^e$ and N^m. In a valid signature, these values will be equal, because $(N^{dm})^e = (N^{de})^m = N^m$

[6] This password-policy mechanism is part of the Krb V5 Admin server software, which OpenVision Technologies wrote and contributed to MIT's Kerberos source-distribution. OpenVision has offices in Cambridge, Mass., Pleasanton, Calif., and London, UK.

[7] The CRL is a bottleneck, but many applications can temporarily waive CRL-checks, during lapses in the CRL's service.

	Public-Key	Symmetric-Key
Adding New Users	bad scaling	bad scaling
Revocation	bad scaling	easy scaling
Out-of-Band Validation	Root Key: hard, frequent	1^{st} PW: easy, only once
Theft Exposure	long-lived key valuable	short-lived key little value
Password-Quality	optional	enforced
Network Bottleneck	CRL service	KDC
Physical Security	client, CA, CRL	KDC
Key-Mgt Responsibility	end-user	sys-admin

Table 1. Compliance defects and administrative differences between public-key and symmetric-key security systems.

too, because the KDC is a symmetric-key network's "single point of failure."

In summary, a public-key security infrastructure has four advantages over a symmetric-key KDC:

- Less trust,
- Lower availability demands,
- Better performance,
- Better reliability

However, these attractive features come at the cost of transferring corresponding burdens onto users. The first such transfer is well-known: public-key cryptography entails a lot of local computation. Poor local performance is the price of avoiding the KDC's bottleneck in network performance and reliability. Essentially, users avoid waiting through one or two seconds of extra network-latency, by spending a comparable period on bignum arithmetic.

The second administrative transfer is the focus of this paper: the public-key infrastructure is less trustful and less available, and hence is easier to administer, but only because the hard part of public-key administration is local and cannot be centralized. Contrary to general belief, public-key cryptography does not abolish administrative trust and diligence. Instead of the users having to trust and rely upon a central organization, every user is responsible for administering his own security. What's worse, an organization's overall security depends primarily on all users to be diligent in their key-management duties.

This reliance on users' diligence is utterly unrealistic. Anderson has collected many case-studies of poor security practices among financial users of symmetric-key security [1]. He consistently found that application-programmers and end-users do not understand, and will not perform, simple key-management duties. Since symmetric-key users have a much lighter key-management burden than public-key security would impose, it is plain that compliance will be the weak link in public-key-secured networks, too.

5 Repair

For a mass-market public-key system to solve these problems, it would have to incorporate highly-available, trusted, and secure servers. Such solutions would add centralized infrastructure costs to public-key's already substantial performance costs. At the cost of introducing administrative trust, symmetric-key systems solve all but the problem of long-distance account-creation, which is hard for any security system.

We can combine both cryptosystems' administrative benefits, by restricting public-key deployment to servers, and by using symmetric-key protocols for desktop clients. [5] The clients' KDC can enforce password-quality, issue short-lived keys, validate servers' public keys, and maintain CRLs. For signatures and asynchronous messaging, a symmetric-key-based signature system can mediate between native symmetric-key users and external public-key users. [13, 6] This hybrid security system would put public key-pairs only in the hands of well-trained sysadmins, and would also minimize the CRLs' scaling problems. Hybridization trades away theoretically perfect privacy, so as to strengthen public-key's practical weak link: user compliance.

Smart-card hardware can repair some compliance defects, but they fall far short of completeness. Smartcards are completely effective only for Root-key validation. For passphrase QA and private-key

management, smartcards just substitute the problem of physical security. Smartcards offer no help for the problems of authenticated issuance and revocation.

6 Conclusion

Compliance defects impede the sound management of keys and of user-accounts. These defects have arisen from the introduction of public-key cryptography into mass-market software As public-key security was originally envisioned during the '70's and '80's, sophisticated users and sensible key-hygiene were taken for granted. For example, Privacy-Enhanced Mail's designers explicitly expected that the professionals who then used e-mail would be able to hand-check their copy of the Root-CA's public key. With the advent of mass-market electronic commerce, this assumption no longer obtains. Nowadays, the security system must be transparent wherever possible, and where transparency fails, it must enforce good key-hygiene.

Public-key cryptography is actually no more "trustless" than symmetric-key security systems. Public-key's decentralized nature actually places a lot of trust on *users*, that properly belongs to the security infrastructure and its administrators. Up to now, this trust in users' discipline has been implicit, and has received little or no attention in discussions of the Internet's security infrastructure. However, it's time to re-assess public-key's "trustlessness," as we approach the large-scale deployment of public-key protocols for electronic commerce.

7 Acknowledgements

Dan Geer, Barry Jaspan, Win Treese, Jon Gossels, Karl Andersen, and Brad Johnson were helpful when I discussed these ideas with them. I also received helpful critique from the USENIX referees.

References

[1] R.J. Anderson, "Why Cryptosystems Fail," *Comm. ACM*, v bf 37 no. 11 (November 1994), pp. 32-40

[2] R.J. Anderson, R.M. Needham, "Robustness Principles for Public-Key Protocols," *Advances in Cryptology - CRYPTO '95*, Springer-Verlag, Berlin, 1995.

[3] M. Burrows, M. Abadi, and R. Needham, "A Logic of Authentication," *Proc. R. Soc. Lond. A* 426(1989) pp. 233-271.

[4] D.A. Curry, *UNIX System Security: A Guide for Users and System Administrators*, Addison-Wesley Professional Computing Series (Reading, Mass.) 1992.

[5] D. Davis, "Kerberos Plus RSA for World Wide Web Security," *Proc. 1st USENIX Workshop on Electronic Commerce* (NYC, 7/95), pp.185-8.

[6] D. Davis and R. Swick, "Network Security via Private-Key Certificates," *USENIX 3rd Security Symposium Proceedings*, (Baltimore; Sept. '92) pp. 239-42. Also in *ACM Operating Systems Review*, v.24, 4 (Oct. 1990).

[7] D. Geer and J. Rochlis, "Network Security: The Kerberos Approach," Usenix Workshop Tutorial.

[8] L. Guillou and J. Quisquater, "A Practical Zero-Knowledge Protocol Fitted to Security Microprocessor Minimizing Both Transmission and Memory." *Advances in Cryptology - EUROCRYPT '88*, Springer-Verlag, Berlin, 1988.

[9] Clyde Hoover wrote the `npasswd` command at U. Texas at Austin: *ftp://emx.utexas.edu/pub/npasswd*. For a concise description, see [4], p. 171.

[10] C. I'Anson and C. Mitchell, "Security Defects in CCITT Recommendation X.509 - The Directory Authentication Framework," *ACM Comp. Comm. Rev.*, (Apr '90), pp. 30-34.

[11] International Telegraph and Telephone Consultative Committee (CCITT). Recommendation X.509: The Directory - Authentication Framework. In *Data Communications Network Directory, Recommendations X.500-X.521*, pp. 48-81. Vol. 8, Fascicle 8.8 of *CCITT Blue Book*. Geneva: International Telecommunication Union, 1989.

[12] C. Kaufman, R. Perlman, and M. Spencer, *Network Security: PRIVATE Communication in a PUBLIC World*, Prentice-Hall Series in Computer Networking and Distributed Systems, (Englewood Cliffs, NJ) 1995, pp. 436-8.

[13] B. Lampson, M. Abadi, M. Burrows, E. Wobber , "Authentication in Distributed Systems: Theory and Practice" *13th ACM Symposium on Operating Systems Principles* pp. 165-182, Oct. 1991

[14] Netscape Communications, "Secure Socket Layer Reference Document," Unofficial Internet Draft.

[15] C. Neuman and J. Kohl, *The Kerberos Network Authentication Service (V5)*, Internet RFC 1510, September 1993.

[16] E. Rescorla and A. Schiffman, "Secure Hypertext Transfer Protocol," Internet Draft RFC, May '95.

[17] R. Rivest, A, Shamir, L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Comm. ACM*, v. 21, **2**, Feb. '78, pp. 120-126.

[18] J.I. Schiller and D. Atkins, "Scaling the Web of Trust: Combining Kerberos and PGP to Provide Large Scale Authentication," *USENIX Winter Conference Proceedings*, January 1995.

[19] E.H. Spafford, "Observing Reusable Password Choices," *USENIX 3rd Security Symposium Proceedings*, (Baltimore; Sept. '92), pp. 299-312.

[20] J.G. Steiner, C. Neuman, and J.I. Schiller, "Kerberos: An Authentication Service for Open Network Systems", *USENIX Winter Conference Proceedings*, February 1988. [athena-dist.mit.edu:pub/kerberos/doc/usenix.PS]

[21] *USPS Electronic Commerce Services*, product information sheet (Washington, DC), 1995.

[22] Visa International and MasterCard, "Secure Electronic Transactions Protocol Specification."

[23] Visa International and Microsoft Corp., "Secure Transaction Technology Specifications."

[24] A. Young, M. Yung, "The Dark Side of Black-Box Cryptography -or- Should We Trust Capstone?" *Advances in Cryptology - CRYPTO '96*, Springer-Verlag, Berlin, 1996.

[25] P. Zimmermann, *The Official PGP User's Guide*, MIT Press (Cambridge, Mass.) 1995.

8 Appendix: Symmetric Key-Distribution

A symmetric Key-Distribution Center is a trusted server that knows each user's password. The KDC issues temporary session-keys to users who know their passwords. Each user's initial session-key comes to him under his password's encryption. The KDC then uses this initial key to encrypt the user's subsequent session keys.

1. Account-Creation:

- The user proves his identity to the KDC's administrator (not electronically).
- The administrator creates an initial password for the user, and tells the user to change it immediately.

2. Single-Sign-On

- At login, the user types his password, so as to decrypt his daily temporary session-key.
- The user applies this session-key in a similar protocol repeatedly through the day as he accesses services, gaining a new session-key for each different server.

3. Authenticating Others

- To communicate securely with other users and with networked services, the user applies various session-keys in a simple protocol.
- In each repetition of this authentication protocol, the KDC identifies the session-key's owners to each other.

4. Password-Change

- The KDC can require the user to change his password regularly, as a condition for access.
- When the user changes his password, the KDC can examine it, so as to enforce complexity criteria on the user's choice.
- The KDC stores the new password in the database, in a hashed form.

5. Account-Revocation

- Session-keys are timestamped to expire quickly, usually after 8 hours or even a few minutes. This discourages key-theft.
- If a user's password is compromised, then he must inform the CRL administrator, who manually replaces the user's password, and tells the user to change it immediately..

The KDC's trusted role gives it potential access to all of the system's traffic. In return, the KDC takes responsibility for managing, validating, and renewing all of the system's keys. Thus, compared to a public-key system's security, all of the KDC's links are weakened somewhat, except for the weakest: the user's key-management link is greatly strengthened.

Texas A&M University Anarchistic Key Authorization (AKA)

David Safford, *Texas A&M University (Dave.Safford@net.tamu.edu)*
David K. Hess, *Texas A&M University, (David-Hess@net.tamu.edu)*
Douglas Lee Schales, *Texas A&M University, (Doug.Schales@net.tamu.edu)*

Abstract:

At the Fourth USENIX Security Conference, we presented a paper on SRA Telnet, which was a simple Diffie-Hellman based system to defeat standard password sniffing, without requiring externally validated keys. Since that time, several projects, such as Secure Telnet (stel), have worked to extend this simple Diffie-Hellman model to add data encryption, larger keys, and resistance to man-in-the-middle attack. Other projects, such as SSL and SSH use validated RSA keys for full authentication. This project uses standard PGP keys as the basis of unified authentication, authorization, and encryption, combining both perfect forward secrecy and strong RSA authentication.

1. Introduction:

AKA has several design goals, including:

- RSA authentication based on an individual's standard PGP key
- Perfect forward secrecy
- Unified, powerful authorization across applications
- Encryption of all authentication and application data
- Immunity to both passive and active attacks
- Full mutual repudiation
- Key security (An individual's secret key never leaves the desktop)
- Distinction between an individual and his account(s)
- Interoperability with unmodified telnet and ftp systems
- Socket layer implementation for ease of incorporation
- Compatibility with connection oriented services

The remainder of the paper presents AKA in five major sections:

- The basic cryptographic protocol
- The authorization model and file format
- The software architecture
- Comparison to other systems
- Summary

II. AKA Basic Cryptographic Protocol:

For AKA, we realized that a good way to provide true mutual authentication without shared secrets was with an RSA based public key system with authenticated public keys. In the past, RSA has been avoided because of the difficulty of providing authenticated key service, and because of the patent status of RSA. Both of these issues have been largely overcome by the release of MIT PGP-2.6.2 [8]. AKA's use of PGP keys provides both essential secure authentication functions; mutual authentication by RSA signing of challenges, and secure encryption of authentication data by RSA encryption with the destination's public key. This system is "anarchistic" in that an individual provides and certifies his own keys, and obtains the server's public key. Neither the user nor the server system need trust any third party Certificate Authority (CA), or Secure DNS service, although these services may be used to obtain keys initially. The AKA protocol provides support for selectable levels of security versus convenience in PGP key initialization.

Perfect forward secrecy means that breaking a secret key will at most compromise one data session; breaking an individual's secret PGP key should not allow decryption of previously recorded sessions. Normally Diffie-Hellman is used with one-time keys to provide perfect forward secrecy, although AKA (and to some extent SSH) use one-time RSA keys in a similar fashion.

The basic design of AKA is very simple. An individual has standard PGP key rings with his protected secret key, his public key, and the public key of a remote service. Similarly the remote service has a secret key, public key, and a copy of the individual's public key. These fixed PGP keys are used only for authentication; separate one-time keys are used to RSA encrypt the AKA exchange to provide perfect forward secrecy. A simple bidirectional random challenge-response exchange is made (to prevent replays from either side), with each item signed with the local side's secret key for authentication, and then encrypted with the other side's one-time public key for confidentiality. (PGP actually signs a hash of the item, to avoid a chosen text attack.) The verified random challenges are then xor'ed to form a session key for a traditional data cipher.

Given a user 'A' wishing to authenticate to remote service 'X', the following high level data flow diagram shows the basic interactions:

```
A                              X
-------- RPKa ------------> (exchange one-time
<------- RPKx ------------- public keys)

------- RPKx(Ra) ---------> (challenge to X)
<----- RPKa(Sx(Ra)) ------- (signed response)

<------ RPKa(Rx) ---------- (challenge to A)
------ RPKx(Sa(Rx)) ------> (signed response)

<--- (Rx xor Ra)(data) ---> (idea/des encrypt
                             all data)
```

where RPKn(R) means RSA encryption of 'R' with one-time public key 'n', Sn(R) means PGP signature of R with secret authentication key 'n'. This basic design has several interesting properties. First, authentication is based on the ability to sign a random challenge with the secret key corresponding to a given public key. The public key signature demonstrates possession of the secret key without exposing it to eavesdroppers, so simple sniffing is ineffective. Since both ends already have the other's public key, neither end can be spoofed. As all exchanges are encrypted with the other side's one-time public key, man-in-the-middle or hijacking is impossible without at least one side's secret key, and perfect forward secrecy is preserved. Authentication based on PGP signatures eliminates the need for any other form of identification, including reusable or one-time passwords, and smartcards.

The use of standard PGP key rings puts control of the keys in the hands of the users, and leverages the key management facilities of the PGP package. An individual's secret key never leaves the local host, and need never be subject to compromise on remote systems.

Addition of Key Initialization Support:

The basic protocol given assumes that the necessary validated authentication keys are already in place. One enhancement to the basic protocol adds an initial key exchange phase to provide the option of key initialization. This modifies the data flow to look like:

```
A                              X
-------- RPKa, PKa--------> (exchange one-time
<------- RPKx, PKx--------- public keys)

------- RPKx(Ra) ---------> (challenge to X)
<----- RPKa(Sx(Ra)) ------- (signed response)

<------ RPKa(Rx) ---------- (challenge to A)
------ RPKx(Sa(Rx)) ------> (signed response)

<--- (Rx xor Ra)(data) ---> (idea/des encrypt
                             all data)
```

Given the availability of unauthenticated keys through this exchange, the client and server side are free to:

- Accept only authenticated keys on existing key rings
- Accept keys if they match those obtained from secure DNS
- Accept transmitted user key based on traditional password (once)
- Accept transmitted keys without validation

The third option provides a convenient method of initializing a user's public key on the server side. In this scenario, a user has (or creates) a PGP key pair on his local machine. The administrator of the remote machine creates a typical user account with traditional password. When the first connection is made through AKA, the provided PKa is conditionally accepted, and installed as a validated PKa only after the user provides his traditional plaintext password. The traditional password is not compromised, as it is transmitted only after the AKA protocol has established the conditionally accepted encrypted stream. In any case, once the validated PKa is installed, the plaintext password will no longer be accepted. All subsequent logins would require full AKA authentication.

The fourth option would provide a similar service to the existing unauthenticated key Diffie-Hellman systems, in which encryption provides only confidentiality to some other authentication system.

Modifications of the protocol for efficiency:

The modified base protocol as presented is still not the final design; it needs to be modified for efficiency. An amazing number of protocols are not organized to parallelize the time consuming encryption operations on both ends of the connection, but rather serialize them. The basic AKA design presented so far, does in fact serialize the encryption operations:

```
            A                  X
            ---- RPKa, PKa---->
            <--- RPKx PKx------
(encrypt)
            ----- RPKx(Ra) --->  (decrypt
                                  sign, and
                                  encrypt)
            <-- RPKa(Sx(Ra)) --
(decrypt &                        (encrypt)
validate)
            <--- RPKa(Rx) -----
(decrypt
Sign and
encrypt)
            --- RPKx(Sa(Rx) -->
                                  (decrypt and
                                  validate)
            <---(Rx xor Ra)--->
```

Here we have 12 RSA operations, requiring 11 sequential time periods. The basic data flow can be reorganized for significantly faster operation:

```
                A                            X
                ---- RPKa, PKa--->
                <--- RPKx PKx------
(encrypt)                               (encrypt)
                ----- RPKx(Ra) --->
                <--- RPKa(Rx) -----
(decrypt                                (decrypt,
sign and                                sign and
encrypt)                                encrypt)

                <-- RPKa(Sx(Ra)) --
                --- RPKx(Sa(Rx) -->
(decrypt &                              (decrypt and
validate)                               validate)

                <---(Rx xor Ra)--->
```

This results in the same 12 RSA operations requiring only 6 time periods.

Another optimization recognizes that creation of 1024 bit ephemeral RSA keys is a time consuming process (around 10 seconds). While it is necessary to use at least one ephemeral key to obtain perfect forward secrecy, it is not necessary for both sides to use one. AKA therefore has the server side pre-generate an ephemeral key, and does not generate on the client side. This way, the initial key exchange need not wait for generation of the ephemeral keys. (SSH uses a similar ephemeral server key, although it generates a new key only once per hour, rather than once per connection, and the ephemeral key is shorter (768 bits). This weakens the perfect forward secrecy, and may allow replay attacks.) Since the challenge Rx from X to A is now encrypted in A's permanent public key, A need not sign Rx to authenticate; simply decrypting PKa(Rx) is sufficient. Therefore A simply returns the hash of Rx to authenticate.

```
A                            X
------ PKa-------->
<--- RPKx PKx------

----- RPKx(Ra) --->
<---- PKa(Rx) -----

<--- PKa(Sx(Ra)) --
---- RPKx(H(Rx) -->

<---(Rx xor Ra)--->
```

Account Considerations:

This section addresses four extensions to the basic model of the identity of the client 'A' and the server 'X':

- separation of principal from account
- default account mapping
- anonymous access
- multiple service keys

A modification to the basic protocol involves support for a separation of the name of the individual (a) on the client from the name of the account (u) he is requesting on the server. While the PGP key identifies (a) with a combination of User ID and Key ID, it does not provide a desired (u), so this information must be sent across. The problem is that we do not want to send this information across before encryption is established, but the information may be needed for the server to validate the provided PKa. For example, the validated key PKa may be stored in the user (u)'s home directory, so the server will need to know (u) before authentication can complete.

Both goals can be achieved by conditionally accepting PKa, and validating the key one step later, as in:

```
A                            X
------ PKa-------->
<--- RPKx PKx------

---- RPKx(Ra,u) -->
<---- PKa(Rx) -----
                             (now validate
                             PKa based on u)
<--- PKa(Sx(Ra)) --
---- RPKx(H(Rx) -->

<---(Rx xor Ra)--->
```

Given potentially multiple mappings from principal to account, a second extension allows for the concept of a "default" account mapping. In this case 'u' may be flagged as requesting the default map, rather than having to name an account specifically.

A third extension is to support the concept of anonymous connections. In this case, a null PKa is sent, the requested 'u' is flagged as anonymous, and the server to client challenge is omitted. The server still uses a combination of authentication and ephemeral keys, although lacking a client key, the server's contribution toward the session key cannot be sent securely to the client, so the client chooses all of the session key. (This is essentially the normal SSH protocol, but with a new ephemeral key per session.)

A fourth consideration for account mapping is to recognize that AKA servers are free (and encouraged) to use a separate service principal for the purpose of selecting the service key. All servers could use a "root" principal and key, for simplicity, but separate principals would be more secure, as the service's secret key must be available to the server in unencrypted form.

III. The authorization model

Existing authorization mechanisms suffer from two significant problems: they are scattered among an unmanageable number of diverse configuration files, and they are frequently based on unreliable information, such as IP address.

The first problem in most systems and especially Unix is the plethora of different mechanisms that exist for service authorization. The following table is a sample of some of the more common mechanisms.

Service	Authorization
many	/etc/inetd.conf
many	/etc/hosts.allow
	/etc/hosts.deny
many	sshd_config
login	/etc/default/login
	/etc/ttytab
login	/etc/skeykeys
r-cmds	~/.rhosts
r-cmds	/etc/hosts.equiv
FTP	/etc/shells
FTP	/etc/ftpusers
FTP	ftpaccess (wuftpd)
FTP,login,POP	/etc/{passwd,shadow}
SMTP	sendmail.cf
	(trusted users)
NFS	/etc/exportfs
	/etc/dfs/dfstab
NFS	nfs_portmon
	(kernel variable)
HTTP	access.conf
lpd	/etc/hosts.lpd
X Windows	xhost
X Windows	.Xauthority
UUCP	USERFILE,L.cmds,...
portmapper	securenets
NTP	ntp.conf
NIS	ypbind (-broadcast)
NIS	ypserv (securenets)
SNMP	community string
TFTP	chroot

The second problem is that traditional network authorization mechanisms typically center around four different types of information: source host, source port, account requested, and service requested. The principal requesting access has to provide a certain combination of these credentials in order to gain access. The source host and source port information is unreliable, and should not be used.

Trying to develop a coherent security policy out of all these mechanisms is daunting.

Rather than introduce yet another authorization mechanism that adds to the mess, AKA tries to define a philosophy and a mechanism that is flexible enough to meet the authorization needs of most services. It is hoped that over time this mechanism can become the standard for user authorization.

There are some key elements to AKA's authorization philosophy. First, AKA discards the philosophy of authenticating via accounts and instead uses principals, where a principal is uniquely identified by the possession of a given secret key with a specified keyid. An individual authenticates a claim to be a given principal by signing a challenge with the principal's secret key. An authenticated principal is allowed to use any account for which it has been authorized.

Also, since the AKA mechanism is based on principals, the concept of including a host and port as part of the authentication mechanism has been removed. Currently this information is unreliable (though IPv6 addresses this problem) and this information does not contribute anything useful to a strong cryptographic authentication of a principal.

The AKA authorization mechanism works with an authorization tuple of (<principal>, <service>, <accounts>, <restrictions>). <principal> corresponds to the entity requesting the service, <service> is the service being requested, <accounts> are the accounts that principal is authorized to use for this service, and <restrictions> are the service specific authorizations (required chroot paths, allowed operations, etc.).

The authorization mechanism defines special tokens for the special case principals of "anonymous", in the unauthenticated case, and "wildcard", to match any (non-anonymous) principal.

All other information pertaining to a service request is considered to be useful for service management, such as the path to the service executable, execution arguments, etc. This information, which is not related to authorization information should be stored in a separate database or file from the authorization information.

IV. The software architecture

A number of software architectures have been used by other projects, such as replacement library (SSL), service wrapper (tcpwrapper [5]), dedicated daemon (sshd), and fully integrated (SRA telnet and ftp). Each approach has certain advantages and disadvantages. The fully integrated approach of SRA has the advantage of full interoperability between modified and unmodified clients and servers, as the use of the service is enabled within the existing application's option negotiation. On the other hand, integration of a new protocol into an existing application is a lot of work, and some applica-

tions may not have existing negotiation mechanisms. Tcpwrapper is the easiest to implement as it requires no modifications to existing executables, but it is limited, as the only authorization information available is based on the IP connection addresses. SSHD has the disadvantage of complete lack of interoperability with existing applications, but this offers it great freedom of implementation. The SSL library approach transparently adds secure socket communications, without modifications to the application code, but does have interoperability problems between modified and unmodified versions.

For AKA, we wanted to satisfy the following architectural goals:

- complete interoperability between modified and unmodified applications.
- little or no modification to application code
- maintain a single point of authorization across all services, whether or not they have been modified.

To satisfy these goals, AKA uses a combination of a replacement socket library and an AKA specific service daemon, which also acts as an ephemeral key service. A client application is linked with the AKA socket library. This library attempts to contact the AKA daemon on the remote server. If there is not an AKA daemon available, the client asks the user if an unsecure connection is ok, and if so attempts to establish a traditional connection.

When a client does successfully contact the AKA daemon, it informs the daemon of the service it is requesting. The AKA daemon then determines how to establish that service. There are three distinct situations that can occur.

The first situation is when the client is attempting to access a service that directly supports AKA, but is not memory resident (it must be started). In this situation, the AKA daemon will behave like **inetd** and will *dup* the socket to the appropriate file descriptors and *exec* the service daemon. The service daemon will request an ephemeral key from the AKA daemon (through a separate channel), and then will execute the AKA protocol. The AKA daemon will be removed from all interaction with the session. This method works when the service daemon is not resident, but in situations where an AKA capable service daemon is itself waiting for network connections, a different technique is necessary.

In this situation, where the service daemon is actively accepting connections, the AKA daemon will not be able to simply use exec to start the service. There are multiple solutions to this. The most efficient is to pass the file descriptor for the session to the service daemon. On many UNIX systems, this can be accomplished through a UNIX domain socket. By passing the file descriptor to the service daemon, the AKA daemon can be removed from having to interact with the session. In situations where this is not possible, the AKA daemon can establish a new connection to the service daemon via a loopback address. The AKA daemon will simply act as a data relayer. The service daemon will have to recognize that connections from loopback are always AKA sessions. When implemented in this manner however, the AKA daemon will remain active in the session, as the data relayer, introducing additional load on the server side.

When the service daemon has not been modified to support AKA, it is still possible to derive some of the benefits of AKA. The AKA daemon will simply connect locally to the service via a loopback connection, and then will sit in front of the service, implementing the AKA protocol. This will introduce additional load on the server side, since an additional process is involved in the communication. Also, the level of AKA support will vary based on the particular service.

Currently the encryption/decryption function will be implemented in either an AKA daemon process, or in the application server process (depending on whether or not the server is AKA modified). It is anticipated that the encryption will eventually be moved into a streams module, or handled by the transport layer (IPv6) for greater efficiency.

V. Comparison to other systems:

Comparison to DES based systems:

Kerberos [4]

In many ways, AKA is the exact opposite of Kerberos. Kerberos depends on a central server with everyone's secret key, while AKA allows users to maintain their own keys in a distributed fashion, with just one copy of the secret key needed on the first machine. Kerberos uses strictly secret key encryption, while AKA uses public keys. Kerberos has difficulties with inter-realm authentication, while AKA does not even have realms.

deslogin [1]

deslogin uses a shared plaintext secret on both ends for one way (user -> server) authentication based on a challenge-response. While it does prevent sniffing and man-in-the-middle attacks, it does not prevent server spoofing, and requires shared plaintext secrets. Since it does not encrypt the full telnet session, it is subject to hijacking or injection.

Comparison to unauthenticated-key Diffie-Hellman systems:

A secure authentication system must do two major things: first it must provide a secure session key for encryption of the service connection, and second it must provide for mutual authentication. Diffie-Hellman, particularly when augmented with the interlock protocol, is a good technique to obtain a common session key. It does nothing, however for authenticating either end of the connection; it simply provides a secure channel over which other authentication methods can be used. The encrypted channel does prevent sniffing, hijack, and man-in-the-middle attacks, but does nothing to prevent client or server spoofing. Authentication of the client (user) to the server must be done with one of the traditional schemes, such as plaintext passwords, or one time passwords (s/key, SecureID...). The problem is that someone could spoof the server to the client and spoof the client to the server, passing the respective challenges and responses between the two separate connections. Diffie-Hellman, even with interlock cannot prevent this "spoof-in-the-middle" form of attack, because it inherently has no way to ensure that it is talking to the right system - it only knows that it is talking securely to whomever is on the other end.

SRA [3]

SRA was a first attempt at a simple Diffie-Hellman based secure authentication system. At the time it was recognized to be susceptible to server spoofing and man-in-the-middle attacks, but these were considered minor compared to the known threat of password sniffing. In addition, its key length was chosen to be compatible with Sun's NIS, which at 192 bits is quite small. Since it does not encrypt the full telnet session, it is subject to hijacking.

Stel [6]

Stel overcomes active attacks, but only at a price; it uses a shared plaintext secret to encrypt the interlock exchange to authenticate both ends to each other, while simultaneously denying man-in-the-middle. Having to have plaintext secrets on all hosts is not very desirable. If you do have them, then why bother with Diffie-Hellman/interlock in the first place? Why not use a simpler and faster challenge/response system directly based on the secret, in a similar (but bidirectional) manner to deslogin?

Comparison to other RSA based systems:

SSL [2]

SSL is based on X.509 (or other) style CA services, which while acceptable for large web based services, is not desirable for individuals, or large numbers of systems, due to the current lack of a standardized CA hierarchy, and the difficulty and expense of registration. In addition, the basic protocol allows the client to choose the session key, which could provide a possible attack. The basic protocol does not have perfect forward secrecy, although version 3 does allow optional Diffie-Hellman. On the other hand, SSL is well suited to the typical secure web model, where many unauthenticated users connect into a relatively small number of certified servers.

SSH [7]

While SSH is similar to AKA in many respects (it is based on individual RSA keys), it does not provide a separation of individual from account, it does not provide a true challenge of the server, it is rsh specific, and does it not interoperate with existing rsh implementations. In addition, its establishment of session key is similar to SSL, in that it allows the client to pick the session key. It attempts to limit key replay exposure to one hour through a periodically changing an ephemeral server public key, but a better approach would be to base the session key on a combination of both client and server information, so that replays were not possible in the first place. The limited duration (one hour) ephemeral key does provide some perfect forward secrecy, although all sessions for the lifetime of this ephemeral key would be vulnerable, rather than the preferred one random key per session. While SSH is very attractive in its ability to forward X11 and other tcp ports, this added power and complexity may open up security holes, and makes SSH difficult to verify.

VI. Summary

AKA provides true mutual authentication that is resistant to sniffing, spoofing, man-in-the-middle, and hijacking attacks, and that does not require shared plaintext secrets. The use of standard PGP keys and key rings makes key management relatively painless. AKA provides fully interoperable services with a single point of authorization for all modified or unmodified services.

In addition to the basic feature of strong distributed authentication, AKA provides some powerful new features:

- Unified authorization.
- Separation of identity from account.
- Key management and security.
- Interoperability with unmodified services.

Unified authorization is an important goal for any security system. One of the problems securing machines is that there are so many different configuration files that must be tailored/checked to ensure security. AKA introduces a unified authorization switch control file, with a unified syntax, to simplify the configuration task. From the application perspective, a single simple API provides all authentication and authorization information through this combined authentication and authorization service.

AKA separates the concepts of identity and host account. An individual is authenticated based on possession of a secret key that matches a given public key. This individual may then be authorized to use one or more host accounts. For example, you do not authenticate the root account, you authenticate "joe", who may in turn be authorized to telnet (or ftp...) into the root account. Most systems disallow directly telnetting to root, instead insisting on logging in as an unprivileged account, and then su'ing to root. AKA allows direct telnet as root, while logging the principal involved.

AKA provides powerful key management and security features. First, AKA uses standard PGP keys, and thus all of PGP's key management facilities. An individual's PGP key pair can be used for both PGP and for all AKA services. An individual's key is entirely under that individual's control, and the secret key need never leave the user's desktop host. Keys are fully distributed - there is no centralized key server.

Unlike SSH, AKA is designed to be implemented to interoperate with existing Telnet and FTP clients and servers. The modified clients and servers negotiate the availability of AKA and can fall back to traditional methods if the other end does not support AKA, and the user/administrator allows. Thus the users need not keep track of which systems support AKA, and use different commands depending on the situation.

Status and Availability:

An initial prototype of AKA is in progress, and further information is expected by the time of the conference. An extension of protocol to handle multi-hop connections is in internal review.

References:

[1] Barrett, D., "Deslogin", ftp://ftp.uu.net:/pub/security/des/deslogin-1.3.tar.gz

[2] Netscape, "SSL Protocol", http://home.netscape.-com/newsref/std/SSL.html

[3] Safford, D., Schales, D., and Hess, D., "Secure RPC Authentication for Telnet and FTP", Proceedings Fourth USENIX Security Symposium, October 1993, Santa Clara, CA, pp 63-67.

[4] Steiner, J., "Kerberos: An Authentication Service for Open Network Systems", Proceedings Usenix Conference, Winter 1988, Dallas TX, pp191-202.

[5] Venema, W., Tcpwrapper, ftp://ftp.win.tue.nl/pub/security/tcp_wrappers_7.4.tar.gz

[6] Vincenzetti, D., "STEL: Secure TELnet", Proceedings Fifth USENIX Security Symposium, June 1995, Salt Lake City, UT, pp 75-83.

[7] Ylonen, T., "SSH", http://www.cs.hut.fi/ssh

[8] Zimmermann, P., "PGP", http://web.mit.edu/network/pgp.html

Murphy's law and computer security

Wietse Venema

Mathematics and Computing Science
Eindhoven University of Technology
The Netherlands
wietse@wzv.win.tue.nl

Abstract

This paper discusses lessons learned from a selection of computer security problems that have surfaced in the recent past, and that are likely to show up again in the future. Examples are taken from security advisories and from unpublished loopholes in the author's own work.

1. Widely-known passwords

Imagine that you choose a password to protect your systems and then advertise that password in big neon signs for all to see. That would not be a very responsible thing to do. Surprisingly, this is almost exactly what some programs have been doing for a long time.

Passwords are the traditional method to authenticate users to computer systems. With todays computer networks, large pseudo-random numbers are being used as password tokens or as cryptographic keys for communication between computer programs. Examples that will be discussed in this paper are Kerberos tickets, X11 magic cookies, and NFS file handles.

These password tokens or cryptographic keys are not chosen by people. Instead, a pseudo-random value is generated programmatically whenever one is needed. Unfortunately it is easy to end up with a predictable result.

1.1. Predictable Kerberos keys

Recent advisories allude to a problem that allows an attacker to impersonate users of the Kerberos version 4 [Stei88, Kohl93] authentication system. In order to understand the problem it is not necessary to go into the details of how Kerberos works. The problem is with the generation of encryption keys.

In the Kerberos system, temporary encryption keys are used to protect authentication information. One of these encryption keys is generated by the authentication server at the very beginning of a login session: it is part of the ticket-granting ticket that allows an authenticated user to obtain service through the network without having to provide a password each time.

Unfortunately, the key generation algorithm used by the version 4 authentication server was predictable. Ultimately, all encryption keys were derived from *known* information (the time of day), from *constant* information (a process ID and a machine identity), and from *predictable* information: a counter that was incremented with each call. The result: a cryptographer's nightmare.

Armed with this knowledge, an intruder could impersonate other users without even having to know their password. The code fragment below illustrates the problem:

```
p = getpid() ^ gethostid();
gettimeofday(&time, (struct timezone *) 0);
/* randomize start */
srandom(time.tv_usec ^ time.tv_sec ^ p ^ n++);
```

The fragment shows a fine example of a comment that lies: feeding predictable input into a deterministic routine produces a predictable result. Curiously, the Kerberos version 4 source code already contains an improved key-generation algorithm that does use secret data as input, but the improved algorithm was not put into actual use until the release of Kerberos version 5[1].

1. Several vendors were already aware of the problem and had taken measures in their own version of the software.

1.2. Only 256 different magic cookies

The X Window system [Schei86, Schei92], comes with the XDM graphical login tool. Some vendors provide their own alternative, but the programs all perform the same basic task: display a logo on the screen and prompt for a login name and a password. When a correct name and password are given, an X session is started. This can be a default desktop that is provided by the system, or it can be a desktop as specified by the commands in a *.xsession* file in the user's home directory.

When an X application program connects to the X server program (i.e., to the user's keyboard, screen and mouse), it typically authenticates according to one of three methods:

- No authentication: every user on the network has access to the user's keyboard, screen and mouse. This is the default on too many systems.
- Client network address: all users on specific hosts have access to the user's keyboard, screen and mouse. The client network address is provided by the client host.
- Magic cookie: all users that know a 128-bit secret value have access to the user's keyboard, screen and mouse. The secret is typically kept in a file *.Xauthority* in the user's home directory. The secret is sent in the clear over the network.

Other authentication methods exist, based on data encryption techniques, but their use is less common. Authentication methods differ in strength. Authentication with magic cookies is more secure than authentication by client network address. Authentication by network address, in turn, is a lot more secure than no authentication at all.

The XDM program suffers from a problem much like the one described earlier for Kerberos version 4: magic cookies are generated from non-secret data (time of day and process ID). Such cookies can be guessed in a small amount of time if one has access to the victim's machine.

Cookie guessing becomes harder, but not impossible, without access to the victim's machine: it is still possible to find out the approximate time of login by fingering the host. However, until recently, some XDM implementations suffered from an even worse flaw that made them vulnerable to arbitrary users on the network. A fix was announced in November 1995.

The problem was than many implementations of the UNIX random number generator are truly horrible: the low 8 bits of its result repeat with a cycle of length 256. These low 8 bits are exactly what some XDM implementations use when they generate a magic cookie:

```
for (i = 0; i < len; i++) {
    value = rand();
    auth[i] = value & 0xff;
}
```

With a cycle of length 256, there can be only 256 different magic cookies! It takes only a fraction of a second to try them all and to find out which of the 256 magic cookies an X server is using. It is as if every other house has the same key to the front door.

Fortunately, many XDM implementations are not vulnerable to this particular problem: they either use a better random number generator or they use an algorithm that involves cryptography.

1.3. Identical NFS file handles

In a discussion of security loopholes, the Network File system [Call95] cannot remain unmentioned. The *fsirand* flaw that I describe here was found and fixed in SunOS 4 years ago. In order to explain the problem I will describe the NFS protocols in a nutshell.

When an NFS client host wants to access a remote file or directory, it sends a request to the file server's NFS daemon. The request includes an NFS file handle that identifies the object being accessed.

How does an NFS client obtain an NFS file handle? Honest clients use the NFS mount protocol. When an NFS client host wants to access a remote file system for the very first time, the client host sends a mount request to the file server's mount daemon. The mount daemon verifies that the client host has permission to access the file system. When the mount daemon grants access, it sends an NFS file handle back to the client.

Once an NFS client has a file handle, it can send file access commands directly to the file server's NFS daemon. In fact, *any client* that is in the possession of a valid NFS file handle can use it. Export restrictions are primarily enforced by the mount daemon. In the most common cases the file server's NFS daemon does not care what client is talking to it.

How, then, does an NFS server protect itself against malicious clients that make up their own NFS file handles? SUN's solution was to make NFS file handles hard to guess. When a file system is created, the *fsirand* program initializes all NFS file handles with pseudo-random numbers. The program is seeded with a process ID and with the time of day. Unfortunately, early fsirand implementations did not initialize the time of day variable. Because of this missing initialization,

the time of day variable contained fixed garbage values, depending on the system architecture. Only the process ID was being set [Dik96].

Many sites run the same *suninstall* procedure to install the operating system onto their disks. This procedure is highly automated, and by implication, the *fsirand* process ID is very predictable. Unknowingly, many sites initialized their file systems with the same NFS file handles world wide.

Thus, in order to access a file system there was no need to use the NFS mount protocol at all. Every other house in the street did have the same keys to the front door.

While researching this paper I noticed that many systems do not even have an *fsirand* command or its moral equivalent. Some systems simply use the time of day when allocating a filesystem inode.

1.4. Moral

Why all this attention to problems with the generation of pseudo-random numbers? The answer is cryptography. Whether we like it or not, cryptography is becoming more and more important for the protection of data and systems. Pseudo-random numbers are essential for the generation of hard-to-guess cryptographic keys. The best encryption in the world is of no use when encryption keys are taken from a predictable source, or when the keys are taken from a too small domain.

In order to generate a secret password you need a secret to begin with. The time of day and the process ID are often not secret. Kerberos is just one system that has suffered from key-generation problems. Another example is the Netscape navigator. Until September 1995, this software generated session keys with only about 30 bits of randomness [Net95]. This amount is even less than the 40 bits that the US government presently allows for exportable cryptography.

In RFC 1750, Eastlake *et al* recommend the use of external sources for randomness [East94]. Even inexpensive computers provide excellent opportunities: for example, keystroke timings, mouse event timings, or disk seek times. The UNIX kernel gives convenient access to all this information and more. By running various incantations of the *ps* command you get a look at information that changes rapidly. When you are attached to a network (and who isn't these days?), statistics from the *netstat* command can be a good source, too, with the caveat that network traffic can be manipulated and monitored. The UNIX kernel is exactly what I used as source of randomness for the SATAN [Farm93] cookie generator: I never even considered the use of a random-number generator.

2. Burning yourself with malicious data

Only a few years ago, Eindhoven University was a peaceful site. Some may remember its name from the days of the great Professor Dijkstra who did fundamental computing science work on structured programming, deadlock avoidance, and so on. And of course, Eindhoven is the place where Philips began making light bulbs and thus started its electronics imperium.

The peace came to an end when Eindhoven University was connected to the global Internet. The university computer systems became immensely popular with data travelers. The facilities had become an excellent starting point to get onto 'the net'. Most visitors were careful not to break things. One visitor, however, had a problem. Every month or so he would break into one of the university computer systems, acquire system privileges, and wipe the machine completely clean:

```
# rm -rf / &
```

The TCP Wrapper [Ven92] began as a simple tool to maintain a log of the intruder's network access attempts. In the course of time I extended the program to learn more and more about the opponent. A powerful extension was the *booby trap*. It allowed us to automatically execute shell commands whenever a suspicious connection attempt was made to our systems. The typical application was to finger the host that connected to our site, in order to get information about the user who was trying to break into our systems.

```
ALL: .bad.domain: \
  finger -l @%h | /usr/ucb/mail root
```

The booby trap feature is very flexible: before the command is given to the shell, it is subjected to substitutions. For example, the sequence *%h* is replaced by the name of the remote host, or by its network address when the name is unavailable.

The TCP Wrapper development process is a tedious one: because so many systems depend on it, everything needs to be tested very extensively. I had been using booby traps for almost a whole year before I included the feature into the public TCP Wrapper release. Only a few months later I received an email message from a kind Mr. Icarus Sparry [Spar92]. He informed me that the booby trap feature could introduce a security loophole.

The problem was as follows. The booby trap feature substitutes host names into shell commands. Host names are looked up via the Domain Name System (DNS), which is a distributed database. The reply to a DNS query can literally come from anywhere on the

Internet. With the booby trap, I was substituting untrusted data into shell commands that were being executed with *root* privileges. This was not nice.

I quickly ran a few tests and, indeed, the DNS server would accept almost anything for a hostname. With an unmodified DNS server I was able to generate hostnames containing various shell metacharacters. In the DNS server data base files, only a few characters have a special meaning (dollar, semicolon, white space and a few others). Anything else can be put into a hostname, for example something as destructive as:

```
>/etc/passwd
```

The attack is not as easy as it seems, though. The TCP Wrapper attempts to expose malicious DNS servers by asking for a second opinion. The program compares the name and address results from a reverse lookup (by host address) with the name and address results from a forward lookup (by host name) and detects discrepancies. Thus, an attacker would have to manipulate the results of both forward and reverse lookups. Nevertheless, it is a bad idea to pass uncensored data from the network into, for example, commands that are given to a shell.

When a program has to defend itself against malicious data, there are two ways to fix the problem: the right fix and the wrong fix. The right fix is to permit only data that is known to give no problems: letters, digits, dots, and a few other symbols. This is the approach that I took with the TCP Wrapper: when doing substitutions on shell commands it replaces characters outside the set of trusted characters by underscores.

Unfortunately, many people choose the wrong fix: they allow everything except the values that are known to give trouble. This approach is an invitation to disaster. Only months ago, part of the WWW (world-wide web) community discovered that CGI (common gateway interface) scripts and other WWW server helper applications can be manipulated by sending data containing newline characters, especially when that data is passed on to shell commands.

2.1. Moral

Recent advisories point out that programs can be manipulated by malicious data from name servers. Together with the *sendmail* program, TCP Wrappers and CGI servers are not the only programs that must defend themselves against malicious data. For example, if you do automated logfile analysis with Swatch [Hans92] or with other tools, you'd better be careful when passing data from logfiles on to other programs: untrusted systems can often send data directly to the *syslog* daemon.

3. Secrets in user-accessible memory

In a previous life I was a nuclear physicist. Working with delicate equipment was a matter of daily routine. Of course things would stop working at the most inconvenient hour of the day. One golden rule that I learned very quickly: if you're fixing a problem, better be sure that you're not breaking something else in the process.

In the information technology world, working with fragile pieces of software is a matter of daily routine. Of course programs stop working at the most inconvenient hour of the day - hardware failures are becoming rare. One golden rule that a system administrator learns very quickly: if you're fixing something, better be sure that the solution does not break something else. Of course we never break something while fixing a problem.

The programmer is faced with the same problems. All too often something breaks while a security or non-security problem is being fixed[2]. Shadow passwords are a good example: they were invented to fix one problem and at the same time opened up a hole.

Shadow passwords attempt to cure a real problem. By the end of the eighties, computers had become fast enough that large-scale password cracking became practical. Alec Muffett's crack program [Muff92] gave everyone the best available tool at the time. Like many UNIX password cracking programs, Alec's program takes encrypted passwords from a password file and tries to guess them in a systematical manner. It is amazing what it found when I first ran it on our own password files.

Some people stated that such programs should not be made available because they might help intruders to break into systems. History repeats. We had a similar discussion again about the release of SATAN, only this time the opposition was much stronger.

Instead of applying censorship to the distribution of software, a more practical solution is to get to the root of the problem: prevent users from choosing weak passwords in the first place, and make encrypted password information less easily accessible. The usual approach is to move the encrypted passwords to a so-called shadow password file that is accessible only to privileged programs. With shadow passwords an attacker can no longer run a password cracker on a stolen copy of the regular password file. Instead, one must connect to the machine to try a password. This makes the risk of detection much larger.

2. Not to mention the things that break as a side effect of fixing a non-problem.

Shadow passwords can give a false sense of security, though. It seems as if there is less need to be careful when choosing a password. After all, the encrypted password is protected from password cracking programs. However, it is easy to see how shadow passwords can actually weaken a system's defenses. Here is a simplified description of how the login program works:

- read username and password from user
- look up password and shadow file entry
- validate username and password
- drop privileges and execute login shell

In the second step, the login program is still privileged, so it can read the secret shadow password file. Every practical login implementation will read a lot more data than just the entry for the user logging in: most likely it reads a whole kilobyte of data or even more. This excess data lingers on in memory buffers somewhere in the process address space.

The fourth step implies a race condition: in-between the time the login program switches to the user and the time it executes the login shell, the login program can be signaled by the user. Shadow password file information that was read in step 2 can be found in the core dump.

It is easy to fall into the trap and keep secret data in the memory of unprivileged programs. I fixed my logdaemon [Ven96] utilities years ago when I ported them to Solaris 2, which has shadow passwords. The fix prevents programs from dumping core.

The SSH [Ylo96] utilities suffered from a similar problem. SSH is a re-implementation of rlogin, rsh and of a few other utilities. It aims to improve host and/or user authentication by the use of strong cryptography. Some of this protection was lost when it was discovered that cryptographic keys remain in memory after a process has switched privilege to the user.

3.1. Moral

The moral of this story is that one should avoid carrying secret information in the memory of unprivileged programs. Preventing core dumps does not give total protection, though; it takes kernel support to protect a process against manipulation with a debugger program[3]. It is therefore essential that a process wipes secrets from memory before switching user privileges. Part of the solution is to use a modified memory manager (malloc) that wipes memory upon return to the free memory pool.

4. Depending on other programs

Being the author of a popular security tool is much like walking a wire thirty feet above the ground without a safety net. The bright side of such an elevated position is that you get a nice view of the world. The dark side is that an error can have painful consequences.

A case in point is the booby trap example of a few sections ago: the example where we finger a host whenever a connection comes from a suspicious source.

```
ALL: .bad.domain: \
  finger -l @%h | /usr/ucb/mail root
```

The booby trap depends on two programs: *finger* and */usr/ucb/mail*. The finger program is relatively harmless: after all, it just connects to the host and reads the reply across the network. Unfortunately, it is not as harmless as it appears to be at first sight. Imagine what happens when the remote finger demon just keeps sending data forever: sooner or later the disk fills up with an enormous temporary file. A judicious link from */dev/zero* to a user's *.plan* file is sufficient to effect a denial of service attack.

The bigger problem is with the dependence on /usr/ucb/mail. This is a complex program with many features. One feature is that it has so-called shell escapes: the mail program recognizes commands in its input when they are preceded by a tilde character. The tilde-command feature was useful when composing a message at the terminal. Nowadays, few people still use the raw /usr/ucb/mail command in interactive mode, but the feature is still there. By inheritance, tilde-command is also supported by the System V *mailx* program.

Three years ago, Borja Marcos pointed out in private email that these tilde commands are also recognized when /usr/ucb/mail reads its input from a pipe [Marc93]. This was bad news: anyone could put shell escape commands into their .plan file and have them executed remotely by triggering a TCP Wrapper finger+mail booby trap.

I decided to silently fix the problem by flooding the market. Each year brought a new major TCP Wrapper release with enough useful features to make people upgrade their old version to the current one. With the *safe_finger* program I attempted to eliminate all known problems with the finger+mail booby trap. While I was

3. In particular, the UNIX kernel should not allow unprivileged users to debug a process with an *effective*, *real*, or *saved* user or group ID that belongs to another user [Ellis95].

at it, I took the opportunity to solve a few other problems, too: fingering a host is not a trivial activity.

This time, at least, I was in good company: a year later it became known that the widely-used INN news transport software [Salz92] had fallen into the same trap. It was possible to execute shell commands world-wide on news servers running INN, by posting just one single news article.

The problem with /usr/ucb/mail shell escapes is going stay with us for quite a while: I have found that many web sites run CGI helper scripts that send data from the network into /usr/ucb/mail, without censoring of, for example, newline characters embedded in the data.

4.1. Moral

Insecurity by depending on other programs is an old problem. The problem becomes even worse when security depends on programs that were not designed for security purposes. Both finger and /usr/ucb/mail are unprivileged programs. They were not designed to defend themselves against malicious inputs from the network. The finger+mail loophole is just one type of accident that can happen. As the author of the TCP Wrapper, it is not possible for me to protect every user against every possible accident. The TCP Wrapper booby trap feature is a sharp tool and it should be used with care.

5. Concluding remarks

The problems discussed in this paper are only the tip of a large ice berg of recurring security problems. There was a lot of material to choose from, and the selection is far from representative. UNIX environment settings are just one example of a major source of recurring trouble that could not be discussed.

All examples in this paper were taken from the UNIX world. Is UNIX such a bad system? Of course not. UNIX is a platform where much pioneering work was done and still is being done. It is therefore not surprising that many errors were made first in the UNIX environment. On the positive side, these experiences should give UNIX users an advantage. For example, now that PC operating systems become capable enough to provide network services, we can expect to see a lot of familiar problems coming back.

As the examples in this paper show, the path of the security programmer is riddled with land mines. The author has had the questionable privilege to meet Mr. Murphy several times in person. Murphy is a cruel teacher.

6. References

[Call95]
B. Callaghan, B. Pawlowski, P. Staubach: NFS version 3 protocol specification. RFC 1813, June 1995.

[Dik96]
Casper H.S. Dik. Private communication, May 1996.

[East94]
D. Eastlake, S. Crocker, J. Schiller: Randomness recommendations for security. RFC 1750, December 1994.

[Ellis95]
James T. Ellis. Private communication, May 1995.

[Farm93]
Dan Farmer, Wietse Venema: Improving the security of your site by breaking into it. ftp.win.tue.nl:/pub/security/admin-guide-to-cracking-101.Z, December 1993.

[Hans92]
Stephen E. Hansen, E. Todd Atkins: Automated system monitoring and notification with Swatch. Proc. UNIX security III, Baltimore, September 1992.

[Kohl93]
J. Kohl, C. Neuman: The Kerberos network authentication service (V5). RFC 1510, September 1993.

[Marc93]
Borja Marcos. Private communication, June 1993.

[Muff92]
Alec D.E. Muffett: Crack version 4.1 - a sensible password checker for UNIX. Part of the Crack distribution, ftp://cert.org/pub/tools/crack/

[Net95]
Netscape technical documents: Potential vulnerability in Netscape products. http://www.-netscape.com/newsref/std/random_seed_-security.html, September 1995.

[Salz92]
Rich Salz: InterNetNews: Usenet transport for Internet sites. Proc. Usenix conference, San Antonio, June 1992.

[Schei86]
Robert W. Scheifler, Jim Gettys: The X window system. ACM transactions on graphics vol. 5, no. 2, April 1986.

[Schei92]
Robert W. Scheifler, Jim Gettys: X window system (third ed.). Digital Press, 1992.

[Spar92]
Icarus Sparry. Private communication, August 1992.

[Stei88]
Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller: Kerberos: an authentication service for open network systems. Proc. winter Usenix conference, Dallas, 1988.

[Ven92]
Wietse Z. Venema: TCP WRAPPER, network monitoring, access control and booby traps. Proc. UNIX security III, Baltimore, September 1992.

[Ven96]
Wietse Z. Venema. The logdaemon utilities provide a login program and several network daemons with enhanced logging and authentication. ftp.win.tue.nl:/pub/security/logdaemon_XX.tar.gz.

[Ylo96]
Tatu Ylönen: SSH - secure login connections over the internet. Proc. UNIX security VI, San Jose, July 1996.

NetKuang – A Multi-Host Configuration Vulnerability Checker*

Dan Zerkle and Karl Levitt
Department of Computer Science
University of California at Davis

zerkle@cs.ucdavis.edu, levitt@cs.ucdavis.edu

Abstract

NetKuang is an extension to Baldwin's SU-Kuang. It runs on networks of computers using Unix and can find vulnerabilities created by poor system configuration. Vulnerabilities are discovered using a backwards goal-based search that is breadth-first on individual hosts and parallel when multiple hosts are checked. An implementation in C++ found real vulnerabilities on production systems. Tests show reasonably fast performance on an LAN.

1 Introduction

The security of modern networked computer systems is dependent on more than just the integrity of the software and protection mechanisms their operating systems use; it is also dependent on the proper configuration and use of that software. Unix computers have a wide variety of security mechanisms such as file permissions, passwords, trusted hosts, and so forth. In practice, such mechanisms can quickly grow very complex. A simple configuration error can lead to users gaining unintended access. The problem has grown worse with the popularity of networked systems. There are more hosts to configure, and the security is dependent on more mechanisms.

This paper presents NetKuang, a system which addresses some of these security concerns by checking networked configurations for unintended security vulnerabilities.

§2 reviews existing systems to enhance Unix security and justifies the need for NetKuang. §3 presents the design of our NetKuang system. It considers the basic functionality of the system, the algorithms used, and the limitations of the design. §4 discusses the current prototype implementation. §5 presents a detailed discussion of the search technique used by the prototype. §6 considers planned enhancements. §7 presents the results of some experiments with this tool. Finally, §8 presents our conclusions about the initial prototype of NetKuang.

*This work is funded by ARPA under Contract No. USNN00014-94-1-0065.

2 Previous Systems

A number of configuration analysis tools have been developed to check whether a system is vulnerable to attack based on the content of system tables. Given that is very likely that an administrator will make mistakes in configuring his system and that many of these mistakes can leave the system open to easy attack, these tools have been widely used as a preventive measure.

The Computer Oracle Password and Security System (COPS) [1], uses a set of shell scripts to check for likely misconfigurations in a Unix system. Among its simple but important checks are the permission modes of security-relevant files and directories, such as `/etc/passwd` and `/etc/group`. COPS also determines if set user-ID root files are world-writable.

The SU-Kuang system [2] is a rule-based expert system for checking the security of a Unix file system's configuration. The rule base captures approaches by which an attacker can extend his privileges by making system calls. Examples of such rule are "if a user can write to a directory, then the user can replace any file inside the directory" and "a user who can replace the password file is able to acquire superuser privileges." Using these rules, SU-Kuang exhaustively searches for all possible actions that enable a user to acquire privileges inconsistent with a specified policy. Although very effective for single host Unix systems, it does not work on a network of Unix systems.

The Miro security constraint file checking system [3] checks a file system against a set of specified con-

straints that define legal configurations of the system. Security constraints are specified as graphs, and the system administrator uses a graphical editor to create and modify graphs. A constraint checker determines if the graphs are consistent with the policy.

Tripwire [4], the widely used file integrity checker, is used to determine if critical files have been modified. An attacker might write trojan horses or trapdoors into critical programs; for example into the `login` program to allow him future access to the system without presentation of password, or into `ls` or `ps` so that calls to these programs will not reveal the presence of modified files or processes. In Tripwire, a cryptographic checksum is computed for each file, and the original checksum is compared periodically with that for the current version of the file.

The Security Administrator's Tool for Analyzing Networks (SATAN) [5] and the Internet Scanner [6] both scan networks to find vulnerable hosts. They can look for such suspicious states as use of faulty versions of network software and improper NFS exports.

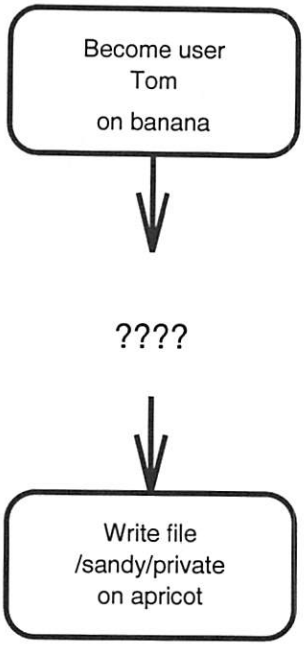


Figure 1: An example query. Given the starting privileges, is it possible to gain the ending privileges?

3 NetKuang

Netkuang is based on Baldwin's SU-Kuang [2], It has all the functionality of SU-Kuang, but also addresses the concerns of a networked environment. It is capable of searching a large number of hosts in parallel, and it also considers potential configuration vulnerabilities present in a networked environment.

NetKuang and SU-Kuang are named after a fictional piece of security-breaking software in William Gibson's novel, *Neuromancer* [7].

3.1 Functionality

An example will best illustrate the kind of problem that NetKuang can find. Consider a user *Sandy*, who is logged into a Unix machine named *apricot*. She wants to find if someone logged into *Tom's* account on host *banana* can potentially modify a file in her home directory called `private`. This possibility is illustrated in Figure 1.

NetKuang might find the following: A user named *Larry* has an account on banana, and his home directory permissions are set to be group-writable. Larry and Tom are both members of the *staff* group, so Tom could replace Larry's shell startup script file. The next time Larry logs in, the modified script makes a copy of the shell. The copy is owned by Larry's account, and has set-user-ID permissions set. Therefore, Tom can log in to banana as Larry.

Another host, *peach*, has an account owned by Larry. Also, peach has a `/etc/hosts.equiv` file that contains banana's name. Therefore, Larry (and now Tom) can log into peach without providing a password. Peach is an NIS server for the network, including apricot and banana. It keeps the NIS database files in a directory called `/var/yp`. The permissions for the `/var/yp` directory are accidentally left world-writable. Therefore, someone using Larry's account can replace the NIS directory with a directory full of modified NIS files. In particular, it is possible to replace the NIS password file there.

Once Tom uses Larry's account on peach to replace the NIS password file, he can give Sandy's account any password he wants. Once this is done, he can use the new password to log in to apricot using the new password. Once logged in, he can use Sandy's account to modify private.txt. This path of privileges is shown in Figure 2.

NetKuang works with privileges as goals. There are only four kinds of goals, as follows: Become (log in as) a user, become a member of a group, write to a file, and replace a file. Each goal is tied to a particular host and has an argument. For example, a fully specified goal is to `write to the file named /etc/passwd on host calvin.comics.com`.

As shown in the example above, privileges of one kind may lead to other privileges. For example, if a user can write to the `/etc` directory on some host, then he can also replace the `/etc/passwd` file on that

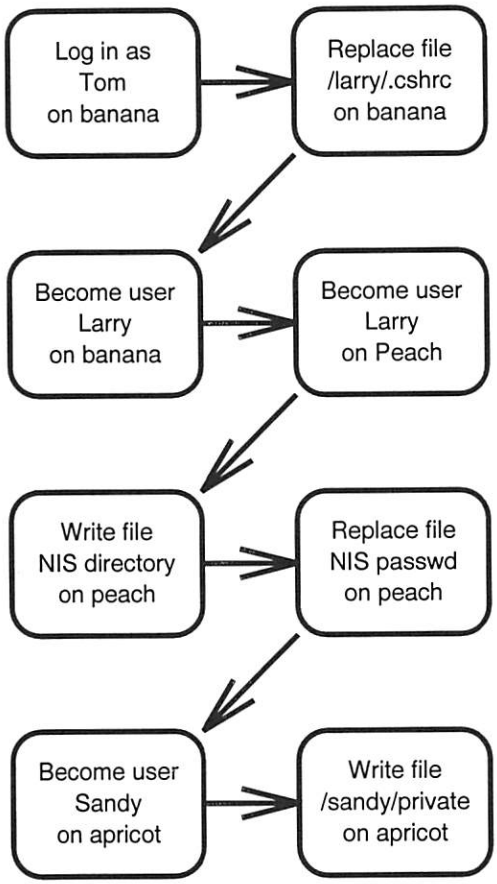


Figure 2: An example path corresponding to the above query.

same host. Doing that will probably lead to further privileges. NetKuang can find long chains of such privileges.

NetKuang, as SU-Kuang, considers such things as file permissions and directory structure. As in SU-Kuang, it also considers the special cases of the permissions on important files such as the password file and users' shell startup files. It has several features beyond SU-Kuang's. It considers the contents of the `/etc/hosts.equiv` file and users' `.rhosts` files to identify which other users can log in without passwords. It also considers the NIS database files, and what privileges may be gained by modifying them. Finally, NetKuang allows wild cards. It can consider all the members of certain groups of users. The set of all rules and goals used by NetKuang is listed in Appendix A.

The user supplies the desired ending goal privileges and the starting privileges that some theoretical misuser might have. NetKuang then performs a search to determine if it is possible to attain the goal privileges from the start privilege. If it finds a solution, it reports the path of privileges leading from the start to the goal.

3.2 Limitations

There are some limitations to the current NetKuang prototype.

It does not consider the integrity of various security-critical software. It does not check for bad passwords or the presence of network sniffers. It does not attempt to parse many important system configuration files, that may have security-critical contents, such as the `/etc/rc` file.

NetKuang only finds a single match. It may be that multiple paths exist between the start and goal privileges. The only way to discover further paths with NetKuang is to fix the known configuration vulnerability and run another search.

4 Current Implementation

The current implentation of NetKuang is written in Gnu C++. It runs on a small network of SunOS 4 and 5 systems. It is currently being ported to IRIX and Ultrix systems on a much larger network for further evaluation.

The search engine is implemented as a daemon on each machine that can be searched. This allows the daemon to directly examine the file system in question. Each engine is capable of participating in multiple simultaneous searches. If a search on a

request fails, the engine may request other engines to carry out the search further. See §5 for a detailed description of the engine's search technique.

The user interface is a simple program that collects the start and destination goals from the user and forwards them to the host identified in the destination goal. Upon completion of the search, the user interface either reports failure or reports the path of goals from the start to the destination. It also notifies the search engines to cancel the completed search.

Communication in NetKuang is carried out by a handcrafted message passing system called ZCS, developed for wide-area auditing. A ZCS message dispatcher daemon collects all NetKuang requests and responses as UDP messages and redistributes them to the modules that need them. Further description of ZCS is not essential for this paper and will be addressed in a future publication.

The search engines must run with super-user privileges in order to examine any `.rhosts` files of users that would not otherwise be readable. If this information is not needed, it may run as any user and unreadable `.rhosts` files are, of course, ignored.

5 Search Technique

NetKuang uses a backward-chained goal-based search. Given starting privileges and ending privileges, NetKuang analyzes the systems on which it is run to determine if the starting privileges are adequate to achieve the ending privileges. The search is carried out breadth-first on each individual host, and in parallel between different hosts. This section details the search technique as it is currently implemented.

5.1 Goals

Searching by NetKuang is based on the generation of goals. Each goal represents privileges that a given user may hypothetically acquire. There are three fields for each goal. The fields are the type of the goal, an argument associated with the type, and the name of the host on which the goal might be met. The different types of goals and their corresponding arguments are listed in Table 1.

Specifying a user by ID and by name are very similar. They simply imply the ability to become the specified user. Unix systems grant most privileges to users based on the value of user ID's. However, the contents of `.rhosts` and `hosts.equiv` files may allow users to log in from remote systems without providing passwords. The identity of remote users

Goal Type	Argument
Become user by ID	User ID number
Become user by name	User name string
Become member of group	Group ID number
Write file	File's path name
Replace file	String file's path name

Table 1: Possible goals and their arguments

is determined by the names of the users, not their ID numbers, so NetKuang tracks both names and ID numbers.

Replacing and writing a file are also similar in that they imply the ability to modify the contents of a file. However, a file must exist before it can be written, while even a non-existent file might be replaced. Also, the ability to write a file implies that the ownership of that file does not change upon modification. Thus, the ability to write a file implies the ability to replace it, but not the other way around. The difference between writing and replacing is important because `.rhosts` files must be owned by the appropriate user or else Unix ignores them when remote users use the `rlogin` protocol.

Here are some example goals:

- `Become user named zerkle on host krakatoa`
- `Become member of group 1 (staff) on host dino`
- `Replace file /etc/passwd on host calvin`
- `Become user ID 0 on host jade`
- `Write file /usr/home/jack/.cshrc on host trusty`

Some goals contain *wild cards*. Goals with wildcards imply special privileges. They are mainly used to specify one complicated search instead of many simple ones.

A goal with a host name of `all` implies the possession of the specified privileges on all hosts searched by a particular instantiation of NetKuang. The host name `any` in a goal implies the possesion of the privileges on at least one searched host.

The two `become user` goals use special wild cards. The user names `all` and `any` imply the ability to become all users or any user at a given host.

Wild cards exist in NetKuang mainly for two particularly interesting goals:

- `Become user all at host all`
- `Become user root at host any`

5.2 Goal Expansion

Goal expansion is the heart of NetKuang's search mechanism. Expanding a goal results in a set of new goals, any one of which, if achieved, grants the original goal privileges. The expansion answers the question, "What are the ways in which I could acquire these privileges?"

Some expansions have preconditions. For instance, the goal `write file` might expand to the ability to become any user in the group of the user that owns the file. The preconditions in this case are that the file exists and that its permissions are set to allow writing by the owner's group.

For instance, the goal

- `become user ID 0 at host apple`

might expand to (among others)

- `replace file /etc/passwd at host apple`,
- `become user named ''root'' at host apple`,
- `write file /.rhosts at host apple`,
- `replace file /var/yp/users/passwd.byuid.pag at host melon`.

As shown by this example, some of the goals resulting from an expansion can be met by the same host as the original goal, but some may be met only by another host.

Expansion of goals with wild cards is handled in a special manner. The goals with the `all` host or user wild card do not expand at all. However, every other goal, in addition to its normal expansions, expands to the same goal with `all` as its host, and every `become user` goal expands to `become user all` at the same host. Every goal, in addition to its normal expansions, expands to the same goal with `all` as its host. A goal with host `any` is expanded to the identical goal for every host on which NetKuang is running, but is not otherwise expanded. The goal of becoming `any` user at a given host expands to one such goal for every user that can log in to that host. All `become user` goals expand to becoming the `all` user on the same host. The goals with the `all` host or user wild card do not expand at all.

A complete list of the possible expansions is given in Appendix A.

5.3 Search Algorithm

A NetKuang search consists of a test of whether a given destination goal can be met from a starting privileges goal. Typically, a search is started by a user request to a search engine.

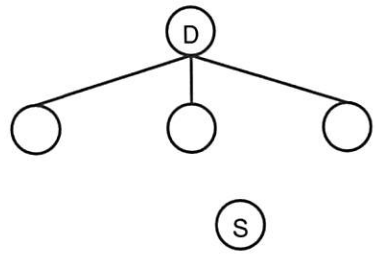


Figure 3: A local search after the first expansion. Node D is the destination goal and node S is the start goal.

NetKuang carries out both local and remote searching. Local searching consists of goal generation and analysis of the local host to determine if the destination goal can be expanded to the origin goal. Remote searching is done if local searching fails to find a solution. It consists of requests to search engines on other hosts to carry out searches to determine if they can meet the goal.

5.3.1 Local Search

Local search is a breadth-first expansion of a goal tree. It is carried out upon receipt of a search request by a search engine, where the request may be made by a user interface or by another search engine. The search request consists of a two goals. One goal identifies start privileges, and the other goal lists destination privileges. Upon receipt of the request, the destination goal is expanded to a new set of goals, as shown in Figure 3. Note that this search is *backwards*. The expansion proceeds from the destination goal in an attempt to find the start goal.

Each of the new goals from the expansion is examined to see if it is the start goal. If not, each of the new goals is expanded. Figure 4 shows these goals partially expanded. One of the new goals is the start goal. This indicates a successful search.

Different goals, when expanded, may result in some identical expansion goals. These might be represented as non-tree links in the search tree, as shown in Figure 5. Such duplicate goals must *not* be further expanded, or loops will result and the search will continue indefinitely. Therefore, the generated goals are kept in a hash table. Newly generated goals already in the table are discarded.

Goals referring to hosts not identical to the host on which the local search is carried out can not be expanded locally. Instead, they are saved for a pos-

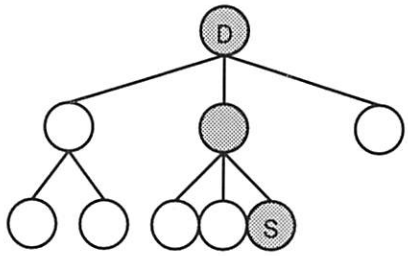


Figure 4: A local search after the second expansion. The start goal has been found. Shaded nodes indicate goals along the successful path.

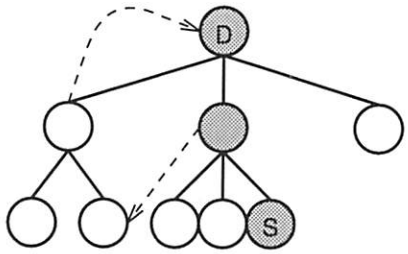


Figure 5: A local search showing non-tree expansions to duplicate goals. The dashed lines indicate the duplicate expansions.

sible remote search.

If the start goal is found, the path between the start and end goals is reported to the search engine or user interface that made the search request. Because the search is breadth-first, it will always find the shortest existing path to the start goal.

If the start goal is not found and no further non-duplicate goals can be generated, the local search is considered to be a failure. If any remote goals are saved from the local search, the search engine will start a remote search.

5.3.2 Remote Search

A search engine carries out a remote search when the local search fails.

For every incoming query, the search engine keeps a set of goals involving remote hosts (actually, there is a queue of goals for each mentioned remote host). When starting the remote search the engine sends one request to each remote host mentioned in a goal generated in the local search. These goals are all sent immediately. The search engine does not wait for results. This means that all the hosts queried can search in parallel, dramatically speeding the search over a serial approach.

If any of the remote requests returns a successful goal path, the engine appends the remote path to the local path used to reach the original remote goal, then returns this successful result to the the user interface or search engine that originated the incoming query.

If a remote request reports a failed search and the incoming query still has remote goals mentioning the reporting host, the search engine will send another query to that host. If no remote goals are left, the engine reports failure to the host that generated the incoming query. This also happens if the local search stage generates no new remote goals.

As with the local search, this recursive remote querying may generate duplicate goals. Therefore, for each ongoing search, each search engine keeps the hash table of previously generated goals. Goals generated from a previous query but the same overall search are ignored in subsequent expansions. A separate hash table is kept for each ongoing search, so the same search engines may be used simultaneously by different searches.

6 Planned Enhancements

Several enhancements are underway to the current implementation of NetKuang.

NetKuang will consider NFS file equivalency. It is possible to modify an NFS-mounted file that is mounted on one host if it is possible to write to the same file on another host. It is quite difficult to keep track of where files are mounted. The planned algorithm is to consider only the host serving a particular remote-mounted file. If the file is local and exportable (the local machine is an NFS server), then the goal of writing the file on a local system expands to writing it on all systems to where it is exported.

Although duplicate searches are avoided on any one particular host, they are not avoided between hosts. If a request included intermediate and failed goals, it would allow hosts to avoid duplication at the expense of higher communication costs. An experiment will be done to determine the performance impact of this feature.

In actual deployment, most search requests are the generic request: "Can any user on any machine get super-user privileges on any machine?" As such, it is a good idea to save results of previous searches and use them to speed up subsequent searches. In fact, the search for super-user privileges could be done on each machine as it starts up its search engine. The partial results are mostly in the form of failed goals which should not be checked again. However, once part of a host's file system changes, some of those goals become invalid and must be discarded.

As any intermediate goal may be generated by multiple paths, it is not clear which such goals should be discarded and which may be kept. An attempt will be made to implement such incremental search.

7 Experimental Results

NetKuang was deployed and run for experimental purposes on a network of ten Sun Sparc workstations. Performance was reasonably fast. It found some real configuration errors.

7.1 Sample Runs

This section details performance tests done with NetKuang. Each test was run five times, and the mean processing time was calculated for each test.

7.1.1 Error on the NIS Server

This test conjectures a configuration error on `olympus`, an NIS server. The error does not actually exist, but the start condition implies it. This demonstrates the "what if" capabilities of NetKuang. Note that the path found was *not* the shortest possible one. The results of a parallel search are nondeterministic, and the first successful goal path is the one reported. The mean run time was 6.2 seconds.

Start condition:

- `Write file /etc at olympus.cs.ucdavis.edu`

End goal:

- `Write file /home/zerkle/tmp at krakatoa.cs.ucdavis.edu`

Goal path:

1. `Write file /etc at olympus.cs.ucdavis.edu`
2. `Replace file /etc at olympus.cs.ucdavis.edu`
3. `Replace file /etc/passwd at olympus.cs.ucdavis.edu`
4. `Become user number 0 (root) at olympus.cs.ucdavis.edu`
5. `Write file /var/yp/seclab.cs/passwd.byuid.pag at olympus.cs.ucdavis.edu`
6. `Replace file /var/yp/seclab.cs/passwd.byuid.pag at olympus.cs.ucdavis.edu`
7. `Become user number 494 (zerkle) at lhotse.cs.ucdavis.edu`
8. `Become user named zerkle at lhotse.cs.ucdavis.edu`
9. `Become user number 494 (zerkle) at krakatoa.cs.ucdavis.edu`
10. `Write file /home/zerkle/tmp at krakatoa.cs.ucdavis.edu`

7.1.2 Can I get root on the server?

This query starts a single-host search on the department server. It is roughly equivalent to an SU-Kuang search, except that it may be initiated from anywhere the NetKuang user interface can be run. Fortunately, the search failed. The mean run time is 1.3 seconds.

Start condition:

- `Become user named ''zerkle'' at toadflax.cs.ucdavis.edu`

End goal:

- `Become user named ''root'' at toadflax.cs.ucdavis.edu`

7.1.3 Can anybody get root anywhere?

This is probably the most standard test. It basically asks if any user on any machine can get root on any machine. The fault found is that the `/etc` directory on lhotse is not owned by root. This run shows a minor limitation of NetKuang. The user interface, running on krakatoa, translated user number 2 to "sys". However, on lhotse, user number 2 is actually "bin". The average run time was 2.9 seconds.

Start condition:

- `Become user named ''all'' at all`

End goal:

- `Become user named ''root'' at any`

Goal path:

1. `Become user named all at all`
2. `Become user named all at lhotse.cs.ucdavis.edu`

3. Become uid ALL USERS (non-root) at lhotse.cs.ucdavis.edu
4. Become user number 2 (sys) at lhotse.cs.ucdavis.edu
5. Write file /etc at lhotse.cs.ucdavis.edu
6. Replace file /etc at lhotse.cs.ucdavis.edu
7. Replace file /etc/passwd at lhotse.cs.ucdavis.edu
8. Become user number 0 (root) at lhotse.cs.ucdavis.edu
9. Become user named root at lhotse.cs.ucdavis.edu
10. Become user named root at any

7.1.4 Can I get root anywhere?

The previous run found a solution too quickly. This run forced the system to check for root-granting configurations on the whole system as a performance check. This search failed. The mean run time was 14.3 seconds.

Start condition:

- Become user named ''zerkle'' at all

End goal:

- Become user named ''root'' at any

7.1.5 Can he log in as me?

This test run is for the sole purpose of demonstrating rlogin privileges granted though the `.rhosts` files. For this run, a vulnerability was purposely introduced to allow user `puketza` on host k6 to log into the `zerkle` account on toadflax. The mean run time was 3.0 seconds.

Start condition:

- Write file /home/puketza/.cshrc at k6

End goal:

- Write file /home/zerkle/tmp at toadflax

Goal path:

1. Write file /home/puketza/.cshrc at k6.cs.ucdavis.edu
2. Replace file /home/puketza/.cshrc at k6.cs.ucdavis.edu
3. Become user number 423 (puketza) at k6.cs.ucdavis.edu
4. Become user named puketza at k6.cs.ucdavis.edu
5. Become user number 494 (zerkle) at toadflax.cs.ucdavis.edu
6. Write file /home/zerkle/tmp at toadflax.cs.ucdavis.edu

7.2 Errors Found

During the testing of NetKuang, some configuration errors were found.

One machine contained several unexpected entries in its t/`.rhosts` file. These entries allow root access to the vulnerable machine to all hosts mentioned in the file.

On the department mail server, the directory containing the NIS database was left world-writable, which means that any user could get root access by replacing the NIS password file.

8 Conclusions

NetKuang is a tool for network administrators that has already demonstrated its usefulness by finding a serious configuration vulnerability. By considering complex configurations and large numbers of hosts, it helps secure modern networks of Unix hosts.

Future plans for NetKuang include integration into an automated intrusion detection system. It can be run periodically to determine if any new vulnerabilities have appeared. If so, they may indicate the presence of an intruder. After an intruder has been detected, it should be run immediately to determine if the intruder has created new vulnerabilities.

NetKuang is under continuing development Complete source and documentation for the latest version of NetKuang is available on the World Wide Web at `http://seclab.cs.ucdavis.edu/~ zerkle/netkuang`.

9 Acknowledgements

Thanks to Todd Heberlein and Calvin Ko for the original idea behind NetKuang.

References

[1] D. Farmer and E. H. Spafford. The cops security checker system. In *Proceedings of the Summer 1990 Usenix Conference*, June 1990.

[2] Robert W. Baldwin. Kuang: Rule-based security checking. Documentation in ftp://ftp.cert.org/pub/tools/cops/1.04/cops.104.tar.

[3] A. Heydon. Specifying and checking unix security constraints. In *Proceedings of the 3rd USENIX Security Symposium*, September 1992.

[4] Gene Kim and E. H. Spafford. The design of a system integrity monitor: Tripwire. Technical Report CSD-TR-93-071, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, November 1993.

[5] D. Farmer and W. Venema. Security administrator's tool for analyzing networks. http://www.fish.com/ zen/satan/satan.html.

[6] Internet Security Systems. Internet scanner. http://www.iss.net/iss/scanner.html.

[7] William Gibson. *Neuromancer*. Ace Books, 1984.

A Search Rules

This appendix lists all of the goal expansion rules used by NetKuang. Unless specified otherwise, the expanded goals refer to the same host as the original goal.

For the purposes of this table, U refers to a user ID number and G refers to a group ID number. A full path name is specified as /d/f, where f is the last component of the path and d is the directory containing f (f may be the name of a directory). The function user(/d/f) refers to the user ID number of owner of the file, while the function group(/d/f) refers to the group ID. On the other hand, uid(name) and gid(name) refer to the user and group ID's associated with user and group names. The uname(U) function reverses this. The expression home(U) refers to the home directory of user U. "Any trusted users" refers to the set of all users specified by user name and host name that are trusted, according to the .rhosts and hosts.equiv files. "Become any user in G" expands to a separate goal for each user in the specified group. These last two may expand to many goals.

Each of these expansions is taken if the precondition is true.

In addition to the expansion in this appendix, goals with wild cards are expanded as described in §5.2.

Replace file /d/f	
Precondition	Expansions
/d/f is not the root directory	replace file /d
(none)	write file /d/f

Table 2: Expansions of **`replace file`** goals.

Write file /d/f	
Precondition	Expansions
/d/f exists	become user(/d/f)
/d/f exists and is group-writable	become member group(/d/f)
/d/f exists and is world-writable	become any user

Table 3: Expansions of **`write file`** goals.

Become user "name"	
Precondition	Expansions
(none)	become uid(name)

Table 4: Expansions of **`become user name`** goals.

Become user-ID U	
Precondition	Expansions
(none)	become uname(U)
(none)	replace file /etc/passwd
(none)	replace password file on NIS server
(none)	become any trusted user
file home(U)/.rhosts exists	write file home(U)/.rhosts
file home(U)/.cshrc exists	replace file home(U)/.cshrc
file home(U)/.login exists	replace file home(U)/.login
file home(U)/.profile exists	replace file home(U)/.profile
U != 0	replace file /etc/hosts.equiv
U == 0	replace file /usr/lib/crontab
U == 0	replace file /usr/lib/aliases
U == 0	replace file /etc/rc
U == 0	replace file /etc/rc.local

Table 5: Expansions of **`become user ID`** goals.

Become group G	
Precondition	Expansions
(none)	replace file /etc/group
(none)	replace group file on NIS server
(none)	become any user in G

Table 6: Expansions of **`become member of group`** goals.

Problem Areas for the IP Security Protocols

Steven M. Bellovin
smb@research.att.com
AT&T Research

Abstract

The Internet Engineering Task Force (IETF) is in the process of adopting standards for IP-layer encryption and authentication (IPSEC). We describe a number of attacks against various versions of these protocols, including confidentiality failures and authentication failures. The implications of these attacks are troubling for the utility of this entire effort.

1 Introduction

The Internet Engineering Task Force (IETF) is in the process of adopting standards for IP-layer encryption and authentication (IPSEC) [Atk95c, Atk95a, Atk95b, MS95, MKS95a]. While these protocols should provide a marked increase in Internet security, they themselves have had a checkered history. It is very much worth recounting the design history, not just to avoid the "oral history" problem in the IPSEC working group, but also because we as a profession learn more from knowing what doesn't work. As a wise sage[1] once said, "Learn from the mistakes of others; you'll never live long enough to make them all yourself."

The failures we discuss here include confidentiality failures—attackers can read encrypted data–and spoofing failures—attackers can transmit phony data. In short, these attacks can render IPSEC useless.

Many (but not all) of the problems stem from the intrinsic properties of the encryption modes used, coupled with the lack of integrity checking in some security transforms and the use of host-pair keying. It has become painfully clear that these combinations are deeply flawed. People assume that since decrypting with the wrong key will yield garbage, additional integrity checking is not needed. Regrettably, this is not the case.

Some of the attacks discussed here were presented informally at the 32nd IETF meeting in Danvers, MA, in March of 1995. Others have been been discussed elsewhere, such as on the IPSEC mailing list or in [WB96].

[1] Alfred E. Neuman of MAD Magazine

2 Properties of Encryption Modes

The ciphers of interest here fall into two broad categories. First, we have block ciphers such as DES [NBS77] used in *cipher block chaining* mode (CBC). Second, the use of stream ciphers has been suggested, in early versions of the SKIP protocol [Azi94] and with the standard ESP header [CW96]; these are byte-at-a-time ciphers. For our purposes, both modes have some significant limitations.

The discussion below focuses on aspects of interest to us. More detailed information on these and other cipher modes can be found in [Sch96].

2.1 Notation

We use $C_i = K[P_i]$ to mean "ciphertext C_i results from the encryption of plaintext P_i using key K. The corresponding decryption is written $P_i = K^{-1}[C_i]$. The symbol $\oplus$ denotes bitwise exclusive-OR.

In showing transforms, the subscript K in ESP_K denotes "ESP encryption using key K".

2.2 Cipher Block Chaining

CBC encryption [NBS80] operates by encrypting the exclusive-OR of each plaintext block and the previous ciphertext block:

$$C_i = K[P_i \oplus C_{i-1}].$$

To encrypt the first plaintext block, C_0 is set to the *initializaton vector* (IV). IVs may be agreed upon in advance, transmitted encrypted, or transmitted in the clear. Using non-constant IVs is often recommended, in order to disguise common prefixes. For our purposes, that does not matter much, as the first encrypted block will almost always be a TCP

[Pos81c], UDP [Pos80], or IP [Pos81b] header, and thus will almost always vary.

Decryption is the inverse operation:

$$P_i = C_{i-1} \oplus K^{-1}[C_i].$$

To encrypt data that is not a multiple of the underlying cipher's block size, some sort of padding and length information must be added. There are a number of different techniques that may be used; none add much to the security of the encryption.

CBC mode encryption has several properties of interest to us. First, the prefix of a CBC encryption is the encryption of the prefix. That is, given a stream of ciphertext $\langle C_1, \ldots, C_i, \ldots, C_n \rangle$, the sequence $\langle C_1, \ldots, C_i \rangle$ is the encryption of $\langle P_1, \ldots, P_i \rangle$. (This may be complicated somewhat by a trailing padding and length indicator scheme.) An attacker can thus truncate a block of encrypted text.

A second property is a generalization of the first. A substring $\langle C_i, \ldots, C_j \rangle$ is a valid CBC encryption of $\langle P_i, \ldots, P_j \rangle$, so long as the IV can be set to C_{i-1}. This allows the attacker to extract any portion of the encrypted message.

The third interesting property is limited error propagation. If a ciphertext block is corrupted in transit, either deliberately or accidentally, only it and the following plaintext block are damaged. If a ciphertext block is dropped, only the following plaintext block will be damaged. The following example illustrates this:

$$\begin{aligned} P_i &= C_{i-1} \oplus K^{-1}[C_i] \\ P_{i+1} &= C_i \oplus K^{-1}[C_{i+1}] \\ P_{i+2} &= C_{i+1} \oplus K^{-1}[C_{i+2}]. \end{aligned}$$

Suppose block C_i is damaged. P_i will be garbled unpredictably, since it depends on the decryption of C_i. P_{i+1} will be damaged in a predictable fashion, since its value is derived from an exclusive-OR with C_i. But P_{i+2} will remain intact, since it depends on C_{i+2} and C_{i+1}, and does not derive from either C_i or P_{i+1}.

Taken collectively, these properties permit cut-and-paste operations. Ciphertext blocks from different messages encrypted with the same key can be combined; only the block immediately following the splice point will be garbled upon decryption.

2.3 Stream Ciphers

There are many ways to build stream ciphers; the ones we are interested in operate by generating a stream of key bytes k_i which are exclusive-ORed with the plaintext, one byte at a time:

$$C_i = k_i \oplus P_i.$$

Clearly, any substring of ciphertext bytes can be decrypted independently, so long as the right starting point is used. There is no error propagation; if a ciphertext byte is damaged, the corresponding plaintext byte is changed in a predictable way, and no other bytes are affected. Stream ciphers of this type cannot cope with byte deletions or insertions.

If the key byte stream is generated by encrypting a counter using a block cipher

$$k_i = K[i]$$

encryption and decryption can start at any point. A standard DES mode, *Output Feedback Mode* (OFB), works by feeding back the block cipher output into itsef:

$$k_i = K[k_{i-1}].$$

Keystreams generated by this mechanism can be cranked forward or backward from a known value of i and k_i, but cannot be started at an arbitrary point.

A third way to generate the key byte stream is by a specialized stream cipher. Whether or not you can restart at an arbitrary point or crank backwards depends on the details of the cipher design.

3 The Attacks

For most of these attacks, we will assume that ESP encryption [Atk95b] is used, but that AH authentication [Atk95a] is not used. Host-pair keying is used. That is, a single key exists between each pair of communicating hosts. This is in distinction to user-oriented keying, or connection-oriented keying, where many keys can be used between two given machines. As needed, we will assume that the attacker X has a legitimate login on one or both of the machines in question, but does not have privileged access to either. Finally, we assume that the attacker has the usual powers over transmitted data: the ability to read, modify, delete, or inject new packets.

3.1 Reading Encrypted Data

The primary purpose of encryption is privacy. An attacker who can read other people's messages has completely defeated the security system. There are a variety of ways in which this can be done.

Assume that a legitimate message is sent from user L_A on machine A to L_B on machine B. The attacker

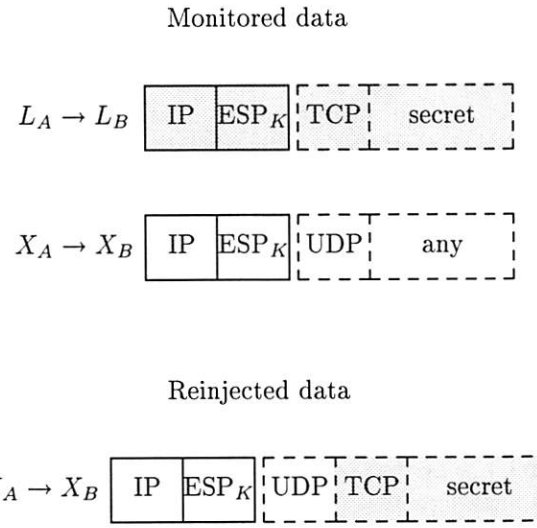


Figure 1: Cutting and pasting legitimate messages to decrypt someone else's traffic. The dashed lines denote encrypted data; the shaded boxes represent data belonging to the legitimate user.

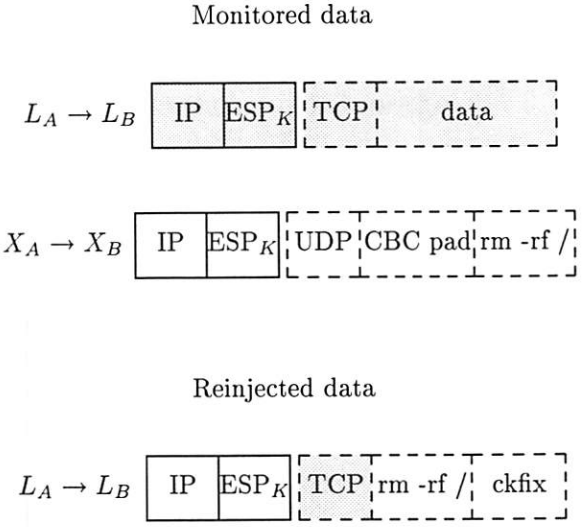


Figure 2: Cutting and pasting legitimate messages to hijack someone else's session. The dashed lines denote encrypted data; the shaded boxes represent data belonging to the legitimate user.

picks this up, and sends a UDP message from X_A to X_B. The encrypted portion of the first message is then inserted into the body of the second message, along with any necessary padding to make the lengths match; this forged message is reinjected into the network for receipt by X_B (Figure 1).

Because of the CBC self-healing properties, the first block of L_A's TCP header may be lost (or may not, if we copy the IV as well). Nothing else will be lost; the body will be readable. If we are using IPv4, the attacker may be able to request that no UDP checksum validation be done; on IPv6, where that isn't possible, on average only about 2^{16} tries are necessary to fool the checksum.

If L_A and L_B are using UDP to communicate, the attack may be even easier. X can wait until process L_B is finished, allocate the same UDP port number on host B as L_B used, and reinject the packets onto the wire where they will be received again by B.. The kernel will decrypt them, and pass them along. The same attack is probably possible with TCP, though it's a bit harder, as TCP's connection-oriented port numbers and sequence numbers will get in the way. Still, on modern high-speed networks it isn't that hard to make the sequence numbers wrap, and the attacker may be able to learn the necessary port numbers by polling via netstat on either machine.

3.2 Session Hijacking

The same sorts of techniques can be used to insert bogus data into someone else's encrypted session. Again, the attacker monitors both a legitimate packet and one sent from X_A to X_B. This second packet contains the data that is to be inserted into someone else's stream. If the legitimate packet can be deleted, the new data can simply be substituted into the payload; if not, a new packet can be constructed with the nasty stuff appended to the legitimate data (Figure 2). After all, TCP has no length field, and it is perfectly content to read a packet where part of the data has already been received but part is new.

Nominally, a padding block must be inserted at the splice point, to prevent the CBC decryption process from damaging valuable spoofed data. The confusion caused by these extra bytes, which will decrypt to garbage, are not significant; the attacker probably needs to send a few extra bytes anyway, to restore to a known state such as the shell prompt.

A related attack involves inducing a machine to send text of the attacker's choice, and cutting it at the proper CBC boundary. The resulting encrypted blocks can be reinjected onto the net, with a new IP header.

Using the chosen plaintext mechanisms discussed below (Section 3.9), it is not even necessary for the attacker to have a login on either machine. The message from X_A to X_B then becomes ciphertext

emitted by host A, in response to X's prompting.

3.3 Fragmentation Attacks

Suppose that an IPSEC implementation does its security processing after fragmentation. Although prohibited by the RFCs, it is comparatively difficult for a "bump-in-the-cord" encryptor to avoid this practice, as the unit may be handed fragments by the host's IP implementation. In that case, the attacker can induce the machine to send a large packet (see Section 3.9), with nasty headers and data just after the fragmentation boundary. The attacker intercepts the packet, changes the IP header to zero the fragment offset field, and reinjects the packet. It will be treated as a complete packet when received. Note that both ESP and AH may be present; the packet will appear to be perfectly valid.

The root cause here is that the fragmentation fields in the IP header are subject to change during transmission, and hence are not included in the AH calculations. This may change for IPv6, where intermediate routers are not allowed to fragment packets. But in that case, the AH header would have to cover the fragmentation header as well. A more general solution is to use tunnel mode for all fragmented packets [WB96]; the inclusion of the extra IP header will protect the fragmentation indicators on the inside.

3.4 Weaknesses of Stream Ciphers

Early versions of the SKIP protocol [Azi94] suggested the use of RC4 [Sch96, pp. 397–398], a stream cipher. Packets included a 64-bit byte sequence number field; the decryptor's stream cipher engine would be cranked until it reached that point. This field also serves as a replay detection mechanism. Authentication was possible but not mandatory.

There are several problems with this scheme. The most obvious is a denial of service attack: an enemy could send a packet with a much larger sequence number, forcing the recipient to spend a lot of cycles turning the crank. In addition, legitimate packets would be rejected, as they would fall between the old sequence number value and the new one—it's difficult or impossible to unwind the cipher state.

The obvious counters to this are to limit the maximum change δ between the sequence number fields in two packets, and to cache recent previous states of the stream cipher engine. But the former runs afoul of the need to span network or host downtime, and the latter can probably be defeated by a burst of forged packets.

Other attacks are more serious. Stream ciphers such as RC4 suffer from a very serious disadvantage: changes to the ciphertext show up as predictable changes to the decrypted plaintext. Suppose that an attacker can trick a machine into sending a known message to a target. This packet can be intercepted, modified, and reinjected onto the wire.

A final attack combines these two threats. Suppose that δ is large enough that an attacker can cause the sequence number field to wrap around. This is difficult but by no means out of the question; if δ is 2^{32}, an enemy who can send at 60% of the bandwidth of an FDDI or 100BaseT net can accomplish this in less than 10 minutes. First cause one machine to send a moderately large amount of known plaintext to the target. This can be done in a variety of ways, such as having it forward a mail message. Next, wrap the sequence number counter. Use the known plaintext to recover the key stream used to protect your text, and use it to encrypt a new message.

A more recent proposal for use of stream ciphers [CW96] addresses some of these issues. For example, a maximum forward change δ is defined, and sequence number wrapping is explicitly prohibited. But this proposal explicitly rejects the inclusion of an integrity check, suggesting that in many cases its use is not necessary, and that an AH header can be added if desired.

3.5 Abusing IVs

Because IVs are sent in the clear, and because after decryption P_1 is produced by an exclusive-OR with the IV, an attacker can introduce predictable changes into P_1. For UDP packets, the entire header is in the first block, thus making it possible to divert packets to new connections. Because of the checksum, changing the TCP headers is somewhat harder, though probably not impossible. IP has a checksum as well, the packet identification field—an arbitrary number—is in P_1; it seems possible to change it to compensate for changes to the the fragmentation control fields; that in turn might enable some of the attacks described in [ZRT95].

Some other attacks on IVs are described in [VK83]; while not all of the scenarios described there are applicable to IPSEC, some are worrisome. But their suggestion that each security association use a separate, secret constant IV does not work well for us; it is too easy to recover most of the bits of the IV. Use a cut-and-paste attack to force decryption of the first block, which will in general be part of either a TCP or an IP header. For the latter, most of the

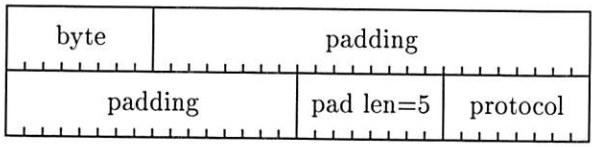


Figure 3: Format of the last block of an ESP packet, when only one data byte is present.

fields are effectively constant, save for the fragment-id which we don't care about in any event; for the latter, the port number fields can be recovered by seeing what ports are in use and the sequence number can be deduced by a modified sequence number guessing attack [Mor85, Bel89]. Any remaining uncertainty can be dealt with by brute force; at most, 2^16 trials will be needed.

3.6 Encypted Hash Functions

It has been suggested that integrity checking can be accomplished by calculating a cryptographic hash of the input packet before encrypting, and encrypting both the packet and the hash. This scheme falls to a cut-and-paste attack using chosen plaintext.

Prepare a new packet, including the hash function output, and cause it to be transmitted. Snip it out of the intercepted packet, using the ciphertext block preceeding the chosen text as the new IV. If the hash function is at the beginning of the packet and a secret IV is used, the mechanisms described above can be employed in conjunction with this scheme.

Other attacks based on integrity-checking failures may be found in [JMM85, SG92a, SG92b, SG93].

3.7 Proxy Encryption

Hosts that will forward received packets that are not addressed to them may be victimized by a proxy encryption attack [WB96]. In this attack, the enemy builds a packet with the IP source address of the forwarder, and a target of some destination machine. If the IPSEC implemenation isn't careful, it will encrypt and authenticate the packet using its own secret keys, thereby convincing the target of the provenance of the message.

Routers are most vulnerable to this attack, since they are in the business of forwarding packets. However, ordinary hosts may be targeted as well. Mechanisms for launching the attack include IP source routing, IP tunneling, and direct injection onto the local network.

3.8 Reading Short Blocks

David Wagner has devised an attack that uses known plaintext and simple active measures to read encrypted data. While the attack is not universally applicable, it does work for the user-to-host data—including any typed passwords—in telnet sessions if either tunnel mode ESP or the TCP timestamp options [BBJ92] are used.

Because DES is a block cipher, data shorter than the block length—64 bits—must be padded to the block length before encryption. The format used for the standard DES-CBC transform is shown in Figure 3. Suppose a single keystroke is being sent. If the preceeding data exactly fills a multiple of this size, this keystroke will occupy the first byte of the last block. The next-to-last byte will contain 5, and the last byte will contain either 6 for TCP or 4 for IP-in-IP. The contents of the intermediate bytes are ignored by the receiving host.

A standard TCP header is 20 bytes long, which is not a multiple of the DES block length. If tunnel mode is used, though, the 20-byte IP header is included, raising the total length to 40 bytes, or eight DES blocks. Similarly, the recommended format for the timestamp options [BBJ92, Appendix A], occupies 12 bytes, bringing the total TCP header length to 32 bytes, which is also an integral number of DES blocks. In either of these cases, single keystrokes will be sent alone in a pad block.

The trick, then, is to send ciphertext blocks whose seventh and eighth bytes of plaintext are the appropriate constants, and whose first byte ranges over the possible character set. If the guessed byte value is incorrect, the TCP checksum will be wrong, and the segment will be silently dropped. A correct value, on the other hand, will elicit an ACK packet, the existence of which (though not, of course, the contents) will be detectable by the attacker. This is true even if the packet represents an old duplicate; the replays will be ignored but will still generate an ACK, according to the TCP specification [Pos81c].

We now need ciphertext blocks with known values for these three bytes; this totals 2^{24} blocks. These could either be generated by the chosen plaintext techniques discussed in Section 3.9, or it could be gleaned by observation. That isn't that hard; if IP tunnel mode is used, the first encrypted block is part of the IP header, and the fields of interest are almost always constant: the IP version, the header length, the type of service, and the fragment offset.

We cannot just use the observed ciphertext for a known plaintext message; the CBC formatting interferes. However, we can recover the appropriate

information. Suppose that C'_i corresponds to plaintext block P'_i encrypted under key K. Then by the rules of CBC decryption,

$$K^{-1}[C'_i] = P'_i \oplus C_{i-1}.$$

Call this value N_i. If the C'_{i-1} values differ, so will the N_i, even if the P'_i values are all the same. This allows us to collect all 2^{24} necessary values. We do not in fact need all possible values of the last two bytes; at this point, however, we do not know which we will need.

Now intercept a encrypted packet $\langle C_1, \ldots, C_n \rangle$ from the target stream. Generate a group of new messages $\langle C_1, \ldots, C_{n-1}, T_t \rangle$ where $T_t = C'_j$ such that $C_{n-1} \oplus N_j$ has t as the first byte and the proper last two bytes. When the receiver decrypts T_k, it sees N_j; stripping off the chaining yields $C_{n-1} \oplus N_j$, which has the proper value. Reinject each of these messages and watch for a 40-byte response. If you see one, you know that t was the encrypted byte.

There are several interesting corollaries to this attack. First, unlike most of our cut-and-paste attacks, the use of AH may not help. If the header sequence is IP-ESP-AH-TCP, the authentication header simply ensures that the TCP portion is correct. But if our guess at t is right, it *will* be correct; we will simply know that twice, once from AH and once from TCP. To be sure, AH failures can trigger alarms, but it isn't clear that this is useful; tearing down a session that received too many AH failures is an invitation to denial-of-service attacks.

Second, the attack is aided because the DES-CBC specification [MKS95a] requires that received padding bytes be ignored. If they had some known fixed value, checked by the receiver, it would be much harder to generate the T_k messages, since far too much known plaintext would be needed. On the other hand, using fixed values will generate plenty of known plaintext for cryptanalytic attacks, every time the user hits ENTER.

A third observation we can make is that using more padding (Figure 4), as is permitted, does not help; in fact, it hurts. We still know that most upstream messages contain a single data byte; if they are the longer, the excess is probably random padding, which means that the block following the header contains only the data byte we are interested in. We don't even have to worry about the padding length and protocol fields, which reduces the known plaintext requirement to 2^8 blocks. In fact, we can use this trick to reduce the total known plaintext requirement to 2^8 blocks!

As before, we build T_t; this time, though, we only care about the first byte t. We now build two more

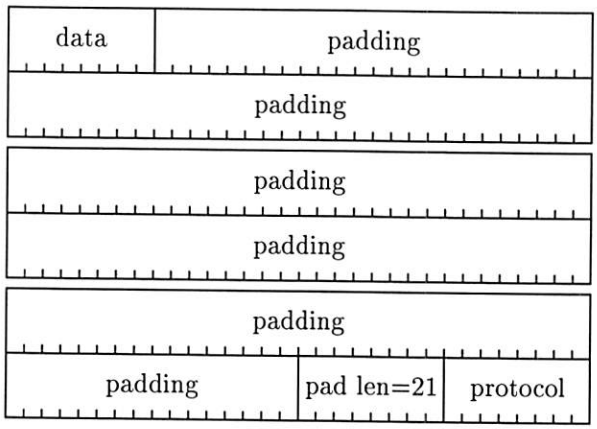


Figure 4: Format of the end of an ESP packet with 21 bytes of padding.

trailing ciphertext blocks, T' and T''. The last block, T'', is a random selection from our table of known plaintexts. But we know what it will decrypt to. The final value comes from the exclusive-OR with T'; we select it so that the final two bytes come out right. We have no idea what the plaintext corresponding to T' will be, but we don't care; it's random padding that the receiver will ignore.

Can we generalize this attack to recover longer sequences than one byte? We may be able to extend it to two byte blocks; beyond that seems unlikely. Collecting the 2^{16} known plaintexts is not too hard; however, we would have to transmit 2^{16} packets containing $T_{t_1 t_2}$.

Going to three bytes creates even more problems; apart from needing to send 2^{24} trial packets, the TCP checksum is only 16 bits; accordingly, there is ambiguity in the decryption. Without good knowledge of the distribution of the t_i bytes, an accurate answer is impossible. Alternatively, one could raise 2^{24} alarms by relying on AH to do the detection.

3.9 Chosen Plaintext

Many of these attacks are aided by how easy it is to make a machine encrypt chosen plaintext. For example, one can often connect to the mail port, and send a long message destined for some user on a neighboring machine. The first machine will happily forward it to the second, permitting the ciphertext to be monitored. To increase the fun, SMTP-level source routing can be done, forcing the first machine to send it to the second, and the second to a non-existent user on a third machine; this last will cause the message to bounce and not be transmitted, but the damage will be done.

Other useful techniques involve exploiting weak-

nesses in the IPSEC implemenation. If, for example, a machine will respond to a plaintext ICMP [Pos81a] or UDP ECHO packet with an encrypted reply, an attacker can forge the initial message and monitor the response. An alternative technique that is often useful is to send an encrypted ECHO message and look for a plaintext response; depending on the local configuration, that may happen as well. We know of several different implementations that will fall victim to one or both of these scenarios.

Beyond their utility for the high-level attacks we have described here, chosen plaintext has cryptographic significance as well. Many cryptanalytic attacks, such as differential cryptanalysis [BS93b, BS91, BS93a], depend on the attacker being able to choose the plaintext to be encrypted. While the quantities needed to attack DES are still out of reach, it appears to be quite feasible to trick local machines into encrypting tens of gigabytes of data. For some ciphers, such as members of the FEAL family [Sch96, pp. 308–311], this may be quite sufficient to mount an attack.

4 Key Changes versus SKIP

A central principle behind SKIP [AMP95] is that a long-lived master key exists implicitly between any two hosts. But this automatic keying makes it difficult for a host to delete a key unilaterally. Recent drafts have addressed this by creating a counter n. If the value of n in a received packet differs from the host's n by more than 1, the packet is rejected. A host can thus delete keys by bumping its counter by 2. Unfortunately, the sender's n is bound to a coarsely synchronized clock, implying that it would not know how to use the new key for up to one hour. Addressing this issue would require long-lived state about the current offset of n on a per-host basis, or easy certificate revocation.

5 Defenses

Against many of these attacks, the proper defense is use of integrity-checking. If a message is properly checked, it cannot be cut apart. More precisely, all received messages should be checked for integrity, using acceptably strong cryptographic techniques. We note that the current protocols do not have a separate mechanism for integrity, as opposed to authentication; however, the authentication transforms do protect the integrity of the message.

A second generic defense technique is to avoid reuse of keying material for more than one "connection". An attacker cannot cut and paste between connections if they use different keys; the inserted material will not decrypt properly.

If this is not feasible, keys should be changed reasonably frequently. For stream ciphers especially, it is necessary to do this based on time, data received, and too large a difference in the indicated sequence number. Recent versions of SKIP [AMP95] have the proper facilities for doing this; it is imperative that older ones not be used.

Replay defenses are also a good idea. If per-connection keying is not used, they are mandatory in certain contexts; packet authentication will not help reject a replay of a perfectly valid packet.

6 Conclusions

The attacks described here are troubling. We have outlined a fair number of very different mechanisms; we strongly suspect there are others as well. Proper cryptographic practice will certainly help; however, there are some very subtle design issues as well, and these are probably harder to find and fix.

In general, hosts should aim for per-connection or per-user keying. The former is probably preferable; the Bad Guys can send evil things to a terminal by way of email, and then replay them to the X server, which would have the same userid. Nor is it always clear who the proper "user" is. Consider the rsh protocol, where a separate connection is set up for stderr. To what userid should this second connection be keyed? Note that while the client program could in principle be running as either the user or root at this point, the server has not been informed of the user's identity yet.

To avoid some of the chosen plaintext attacks, we suggest a simple security policy: never reply to a plaintext message with an encrypted one, and vice versa. If necessary, an ICMP error message can be sent, or key negotiation commenced. Furthermore, SPI pairs should be established by the key negotiation process; messages received via one SPI should always be replied to using its peer. This puts certain constraints on the key management process, especially during rekeying.

It is quite clear that encryption without integrity checking is all but useless. We strongly recommend that all systems mandate joint use of the two options. It is in some sense irrelevant if AH and ESP are two separate protocols, or if integrity checking is made an integral part of each ESP transform; however, the bookkeeping issues may be considerably simplified if the latter path is chosen. Consider, for example, the problem of an authentication key expiring independently of the associated encryption key,

or inconsistent pairwise relationships for AH and ESP. Furthermore, not all combinations are secure; given the way authentication failures can be used to compromise secrecy, the authentication transform must be at least as strong as the secrecy mechanism. It would seem to make little sense to combine MD5 [Riv92], with its $O(2^{64})$ strength against birthday attacks, with triple-DES [MKS95b].

These attacks and recommendations, taken *in toto*, leave us feeling very nervous about network layer encryption. It may be that its promise of transparent, ubiquitous security cannot be kept, at least in general. Use of it when outsiders have access to the endpoint machines, via either logins or network services, seems particularly inadvisable. We suggest that its use be restricted to the following situations:

1. Router-to-router encryption to provide virtual private networks, in conjunction with a firewall. Insiders have other means of attack; the firewall should keep outsiders from mounting chosen plaintext attacks on inside machines. There are still some risks—mail destined for a machine inside one private cloud could be routed by the enemy to the mail gateway inside another cloud; the traffic, when relayed, will be encrypted. There are implications here for the proper integration of encryption and firewalls; we will not pursue the matter further here.

2. As a special case of the above, a "call home" tunnel from a mobile machine to its firewall. A great deal of care must be taken, though, to ensure that the mobile machine does not respond at all to packets sent to its outside address.

3. Possibly between two single-user hosts, though the potential for attack here is quite high.

For more general use, we recommend moving towards the cryptographic processing towards the transport layer. The semantics are quite clear for TCP: for each new socket, create a new pair of SPIs. For UDP, the binding must be between a socket and every host it has ever communicated with. In either case, when the socket is destroyed all of its associated SPIs must be destroyed as well. Looked at another way, the incoming SPI *is* a pointer to the socket. (A useful side-effect of this policy is that ICMP messages will work again: the returned portion of the packet will contain the SPI, which points to the socket.)

A scheme like this could put a heavy load on the key management protocol. Even a simple set of rekey messages would add several round trips; for short exchanges, such as HTTP transfers, this is probably unacceptable. We suggest that the key exchange protocol allocate n SPIs, n probably a power of 2, where the key for SPI_i is some one-way function of i and the negotiated master key. A host could start using a new SPI in the allocated range without further negotiation; to avoid race conditions, the initiator should start allocating from one end while the responder allocates from the other. A new group of SPIs would be allocated when too few were left in the old group.

The situation is rather more problematic for things like DNS servers, which would have to maintain very many active keys. SKIP would simplify the situation, as each message could have its own traffic key K_p; however, it suffers from the flaw that a receiver has no way to force the sender to use a different key. Probably, the right answer is to omit IPSEC entirely for DNS messages, and instead rely on authenticated DNS records [EK96].

7 Acknowledgments

Some of these specific attacks were first pointed out by others, including Ashar Aziz of Sun Microsystems and David A. Wagner of the University of California at Berkeley. Wagner, Ran Atkinson, Perry Metzger, and Hilarie Orman made many useful suggestions about draft versions of this paper.

References

[AMP95] Ashar Aziz, Tom Markson, and Hemma Prafullchandra. Simple key-management for Internet protocols (SKIP). Internet draft; work in progress, December 21, 1995. (draft-ietf-ipsec-skip-06.txt).

[Atk95a] R. Atkinson. IP authentication header. Request for Comments (Proposed Standard) RFC 1826, Internet Engineering Task Force, August 1995.

[Atk95b] R. Atkinson. IP encapsulating security payload (ESP). Request for Comments (Proposed Standard) RFC 1827, Internet Engineering Task Force, August 1995.

[Atk95c] R. Atkinson. Security architecture for the internet protocol. Request for Comments (Proposed Standard) RFC 1825, Internet Engineering Task Force, August 1995.

[Azi94] Ashar Aziz. Simple key-management for Internet protocols (SKIP). Obsolete Internet draft, October 25, 1994. (draft-ietf-ipsec-aziz-skip-00.txt).

[BBJ92] D. Borman, R. Braden, and V. Jacobson. TCP extensions for high performance. Request for Comments (Proposed Standard) RFC 1323, Internet Engineering Task Force, May 1992. (Obsoletes RFC1185).

[Bel89] Steven M. Bellovin. Security problems in the TCP/IP protocol suite. *Computer Communications Review*, 19(2):32–48, April 1989.

[BS91] Eli Biham and Adi Shamir. Differential cryptanalysis of DES-like cryptosystems. *Journal of Cryptology*, 4(1):3–72, 1991.

[BS93a] Eli Biham and Adi Shamir. *Differential Cryptanalysis of the Data Encryption Standard.* Springer-Verlag, Berlin, 1993.

[BS93b] Eli Biham and Adi Shamir. Differential cryptanalysis of the full 16-round DES. In *Advances in Cryptology: Proceedings of CRYPTO '92*, pages 487–496. Springer-Verlag, 1993.

[CW96] Germano Caronni and Marcel Waldvogel. The ESP stream transform. Internet draft; work in progress, April 1996. (draft-caronni-esp-stream-00.txt).

[EK96] Donald E. Eastlake, 3rd and Charles W. Kaufman. Domain name system protocol security extensions. Internet draft; work in progress, January 30, 1996. (draft-ietf-dnssec-secext-09.txt).

[JMM85] Robert R. Jueneman, Stephan M. Matyas, and Carl H. Meyer. Message authentication. *IEEE Communications*, 23(9):29–40, September 1985.

[MKS95a] P. Metzger, P. Karn, and W. Simpson. The ESP DES-CBC transform. Request for Comments (Proposed Standard) RFC 1829, Internet Engineering Task Force, August 1995.

[MKS95b] P. Metzger, P. Karn, and W. Simpson. The ESP triple DES-CBC transform. Request for Comments (Experimental) RFC 1851, Internet Engineering Task Force, October 1995.

[Mor85] Robert T. Morris. A weakness in the 4.2BSD UNIX TCP/IP software. Computing Science Technical Report 117, AT&T Bell Laboratories, Murray Hill, NJ, February 1985.

[MS95] P. Metzger and W. Simpson. IP authentication using keyed MD5. Request for Comments (Proposed Standard) RFC 1828, Internet Engineering Task Force, August 1995.

[NBS77] NBS. Data encryption standard, January 1977. Federal Information Processing Standards Publication 46.

[NBS80] NBS. DES modes of operation, December 1980. Federal Information Processing Standards Publication 81.

[Pos80] J. Postel. User datagram protocol. Request for Comments (Standard) STD 6, RFC 768, Internet Engineering Task Force, August 1980.

[Pos81a] J. Postel. Internet control message protocol. Request for Comments (Standard) STD 5, RFC 792, Internet Engineering Task Force, September 1981. (Obsoletes RFC0777).

[Pos81b] J. Postel. Internet protocol. Request for Comments (Standard) RFC 791, Internet Engineering Task Force, September 1981. (Obsoletes RFC0760).

[Pos81c] J. Postel. Transmission control protocol. Request for Comments (Standard) STD 7, RFC 793, Internet Engineering Task Force, September 1981.

[Riv92] R. Rivest. The MD5 message-digest algorithm. Request for Comments (Informational) RFC 1321, Internet Engineering Task Force, April 1992.

[Sch96] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* John Wiley & Sons, New York, second edition, 1996.

[SG92a] Stuart G. Stubblebine and Virgil D. Gligor. On message integrity in cryptographic protocols. In *Proc. IEEE Computer Society Symposium on Research in Security and Privacy*, pages 85–104, Oakland, CA, May 1992.

[SG92b] Stuart G. Stubblebine and Virgil D. Gligor. On message integrity in cryptographic protocols. Computer Science Technical Report 2843, University of Maryland, College Park, MD, February 1992.

[SG93] Stuart G. Stubblebine and Virgil D. Gligor. Protocol design for integrity protection. In *Proc. IEEE Computer Society Symposium on Research in Security and Privacy*, pages 41–53, Oakland, CA, May 1993.

[VK83] V. L. Voydock and S. T. Kent. Security mechanisms in high-level network protocols. *ACM Computing Surveys*, 15(2):135–171, June 1983.

[WB96] David A. Wagner and Steven M. Bellovin. A "bump in the stack" encryptor for MS-DOS systems. In *Proceedings of the Symposium on Network and Distributed System Security*, pages 155–160, San Diego, February 1996.

[ZRT95] P. Ziemba, D. Reed, and P. Traina. Security considerations for IP fragment filtering. Request for Comments (Informational) RFC 1858, Internet Engineering Task Force, October 1995.

NOTES

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of engineers, scientists, and technicians working on the cutting edge of the computing world. The USENIX conferences and technical symposia have become the essential meeting grounds for the presentation and discussion of the most advanced information on the developments of all aspects of computing systems.

USENIX and its members are dedicated to:

- problem-solving with a practical bias,
- fostering innovation and research that works,
- communicating rapidly the results of both research and innovation,
- providing a neutral forum for the exercise of critical thought and the airing of technical issues.

USENIX holds an annual technical conference, an annual system administration conference co-sponsored with SAGE, and single topic symposia throughout the year. It publishes *;login:*, a bi-monthly newsletter; *Computing Systems*, a quarterly technical journal published in association with The MIT Press; and conference proceedings for each of its conferences and symposia. It also sponsors local and special technical groups relevant to the UNIX environment as well as participating in various standards efforts.

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX. SAGE activities include the publishing of *Job Descriptions for System Administrators*, edited by Tina Darmohray; "SAGE News", a regular feature in *;login:*, The System Administrator Profile, an annual survey of system administrator salaries and responsibilities; support of working groups; and an archive site for papers from the System Administration Conferences.

Member Benefits:

- Free subscription to *;login:*, the Association's bi-monthly newsletter featuring technical articles, a worldwide calendar of events, SAGE News, media reviews, summaries of conferences, Snitch Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts, and much more.
- Free subscription to *Computing Systems*, a refereed technical quarterly published with The MIT Press
- Discounts on registration for technical sessions at all USENIX conferences and symposia
- Discounts on proceedings from USENIX conferences and symposia
- Discount on the new 4.4BSD Manuals, the definitive release of the Berkeley version of UNIX with a CD-ROM published by the USENIX Association and O'Reilly & Associates, Inc.
- Savings on *The Evolution of C++: Language Design in the Marketplace of Ideas*, edited by Jim Waldo of Sun Microsystems Laboratories, the USENIX Association book published by The MIT Press
- Special subscription rates to *UniForum Monthly, UniNews* and the annual *UniForum Open Systems Products Directory*
- Savings on selected titles from McGraw-Hill, The MIT Press, Prentice Hall, John Wiley & Sons, O'Reilly and Associates, and UniForum

Supporting Members of the USENIX Association:

ANDATACO
Apunix Computer Services
Frame Technology Corporation
ISG Technologies, Inc.
Matsushita Electrical Industrial Co., Ltd.
Motorola Research & Development
Open Market, Inc.
Shiva Corporation
Sun Microsystems, Inc., Sunsoft Network Products
Sybase, Inc.
Tandem Computers, Inc.
UUNET Technologies, Inc.

SAGE Supporting Members:

Bluestone, Inc.
Enterprise Systems Management Corporation
Great Circle Associates
Pencom Systems Inc.
Southwestern Bell

For further information about membership, conferences or publications, contact: The USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA Phone: +1-510-528-8649 Fax: +1-510-548-5738
Email: *office@usenix.org* URL: *http://www.usenix.org*

ISBN 1-880446-79-0